0. Getting started

It is easier to learn *vi* by editing an already existing text file than it is to actually create a new one. Consequently, it may be best to either start editing an already existing C++ source file, or perhaps you may wish to download a text file at Project Gutenberg; for example, the plain-text version of Romeo and Juliet or A Tale of Two Cities. You could also download an HTML source file, as well. Suppose you have some C++ code that looks implements the binomial coefficients. Suppose that is stored in the file binomial.cpp and you type at the command line

```
vi binomial.cpp
```

or

```
vim binomial.cpp
```

you will see a screen similar to:

```
#include <iostream>

// Function delcarations
int main();
long binomial( long n, long k );

// Function definitions
long binomial( long n, long k ) {
        if ( (k == 0) || (k == n) ) {
                return 1;
        } else {
                return binomial( n - 1, k ) + binomial( n - 1, k - 1 );
        }
}

int main() {
        int n{10};

        for ( int k{0}; k <= n; ++k ) {
                std::cout << " " << binomial( n, k );
        }

        return 0;
}
~
~
~
~
"binomial.cpp" 24 lines, 407 characters
```

The source code is visible, and to mark all lines beyond the end of the file, you see a number of tildes (~). The last line of the screen is always for information or various modes. Now, you can see the name of the file, the number of lines and the total number of characters. The first character is highlighted, indicating the position of the cursor.

1. Basic editing

When you open a text file, you are immediately placed into *command mode*. When in *command mode*, the cursor is on one character in the text file. The *current line* will refer to the line on which the cursor is currently located.

`h j k l`	Movement the cursor left, down, up and right.
`x`	Delete the character on which the cursor is sitting (Delete).
`X`	Delete the character immediately before the cursor (Backspace).
`u`	Undo the last edit.
Ctrl-`r`	Redo the last edit that was undone.
`0`	Move the cursor to the first character in a line.
`$`	Move the cursor to the last character in a line.
`a`	Go into *insert mode* after the character on which the cursor is sitting.
`A`	Go into *insert mode* at the end of the current line.
`i`	Go into *insert mode* before the character on which the cursor is sitting.
`I`	Go into *insert mode* at the start of the current line.
Ctrl-`[`	Return from *insert mode* to *command mode*.
`:`	Go to *escape mode*.
`:`*n*Enter	Go to line *n* where *n* is an integer (`:0` is the same as `:1`) and return to *command mode*
`:w`Enter	Save (write) the file and return to *command mode*.
`:q`Enter	Quit.
`:wq`Enter	Save (write) and quit.
`:q!`Enter	Quit without saving.

If you press `:` and enter *escape mode* but do not want to enter any commands, just press Ctrl-`[` or Enter, either of which will return you to *command mode*. If you have partially entered a command in escape mode and do not want to execute that command, you can use Ctrl-`[` to return to command mode.

When you are in *insert mode*, you can only add new characters or delete the most recent character you added by pressing Backspace. If you want to make any more significant changes, you must return to *command mode* by pressing Ctrl-`[`.

For `h`, `j`, `k`, `l`, `x`, `X`, `u` and Ctrl-`r`, typing a non-zero number *n* first and then pressing the key will repeat that operation *n* times. For example, `5j` moves the cursor five lines down.

For `a` and `i`, typing a non-zero number *n* first and then pressing the key will allow you to enter *insert mode* but when you return to *command mode*, that insertion will be repeated *n* times.

2. Searching

When in *command mode*, you can initiate a search by going into *search mode* which allows you to enter text you wish to search for.

/*text*Enter	Move the cursor forward to the next instance of the characters *text*. If the search reaches the end of the file, the search will continue at the beginning. If *text* is not found, *vi* will indicate `Pattern not found: text`
?*text*Enter	Move the cursor backward to the previous instance of the characters *text*. If the search reaches the beginning of the file, the search will continue at the end. If *text* is not found, *vi* will indicate `Pattern not found: text`
/Enter	Repeat the most recent search in the forward direction.
?Enter	Repeat the most recent search in the backward direction.
n	Repeat the most recent search in the same direction of the most recent search.
N	Repeat the most recent search in the opposite direction of the most recent search.

If you press / or ? and enter *search mode* but do not want to search for something, just press Ctrl-[which will return you to *command mode*.

For all of these searches, typing a non-zero number *n* first and will search for the n^{th} next or previous instance of what is being searched for.

3. Search-and-replace

When in *command mode*, you can enter *escape mode* to perform a search and replace. The format is

```
:s/search-text/replace-text
```

This performs the search-and-replace at most once on the current line. When the search-and-replace is finished, details on how many changes were made are printed to the screen, and an error message is displayed if no matches were found. As you may suspect any `/` in the searched-for or replaced text must be escaped with a backslash `\/`.

You can add additional options, including a range of lines on which to perform the search:

```
:ranges/search-text/replace-text
```

where *range* is one of many possible variations such as

`%`	The entire document.
n	Line *n*.
`.`	The current line.
`.-`*m*	The *m*th line previous to the current line.
`.+`*n*	The *n*th line after the current line.
m`,`*n*	Line *m* to line *n*.
`.-`*m*`,.+`*n*	The *m*th line previous to the current line to the *n*th line after the current line.
`0,.`	The start of the document to the current line.
`.,$`	The current line to the end of the document.
`/`*match-text*`/`	The next line that matches the text *match-text*.
`?`*match-text*`?`	The previous line that matches the text *match-text*.
`g/`*match-text*`/`	Every line that matches the string text *match-text*.

This is not exhaustive; for example, the range `.-5,$` makes a substitution on all lines from five lines before the current line up to the end of the file. There are other means of specifying ranges beyond the scope of this introduction.

There are also options for searches:

```
:ranges/search-text/replace-text/opts
```

where *opts* may be one or more of the following characters:

`g`	The replacement should occur with all matches in the line, not just the first.	
`e`	If no matches are made, do no display the usual error message.	
`i`	Ignore case when making matches.	
`c`	Confirm each change with:	
	`y`	yes, make the change
	`n`	no, don't make the change
	`a`	yes, and do so for all subsequent matches
	`q`	quit the search and replace
	Ctrl-`e`	expose one more line at the bottom of the screen (scrolling down)
	Ctrl-`y`	exposes one more line at the top of the screen (scrolling up)

4. Advanced navigation and searches

You should probably not memorize these advanced navigation and searches. Instead, review them, and later when you begin editing larger files, you will likely find yourself thinking that the basic navigation or searches are too tedious or coarse, and you will then rediscover many of the more efficient means of navigating or searching your file. Needing and then using a feature is a much better and memorable means of remembering that feature than just reading about them.

For the purposes of *vi*, a *word* is any contiguous (touching or adjacent) sequence of either

1. alphanumeric characters (letters or numbers) or the _
2. non-alphanumeric characters with the exception of the _.

The different *word*s are in this line (where Tab indicates a tab character) are highlighted in alternating colors:

```
as_39  2  Tab(#+_32>hello    my_name3+is  Bob!
```

A *Word* is any contiguous sequence of non-whitespace characters. The different *Word*s in this line are

```
as_39  2  Tab(#+_32>hello    my_name3+is  Bob!
```

A *delimeter* is a pair of (,), [,], {, }, < or > used to block text. *vi* will make intelligent guesses as to when < or > is being used as a delimeter and when they are used as less- or greater-than signs.

An *empty* line is any line containing no characters, not even whitespace.

These navigation commands have different meanings if they are prefixed by an integer, and thus are listed separately:

Ctrl-u	Jump backward (up) by half a screen but leaving the cursor on the same line on the screen.
*n*Ctrl-u	Jump backward (up) by *n* lines but leaving the cursor on the same line on the screen.
Ctrl-d	Jump forward (down) by half a screen but leaving the cursor on the same line on the screen.
*n*Ctrl-d	Jump forward (down) by *n* lines but leaving the cursor on the same line on the screen.
H M L	Move the cursor to the first character of the first, middle, or last line of the screen.
*n*H *n*L	Move the cursor to the n^{th} line down from the top or up from the bottom of the screen.
gg	Move the first line of the file.
G	Move the last line of the file.
*n*gg or *n*G	Move line *n* (similar to :*n*Ctrl)
%	Move to the matching delimiter of the current location of the cursor if the cursor is on a delimiter or the next delimiter found moving forward.
n%	This author has not been able to deduce exactly what this does…

For all of these navigation commands or searches, entering a non-zero integer *n* before executing the navigation or search will repeat that navigation or search *n* times.

Enter	Move one line down.
Ctrl-b	Jump backward (up) by a full screen, placing the cursor on the last line of the new screen.
Ctrl-f	Jump forward (down) by a full screen, placing the cursor on the first line of the new screen.
w	Move the cursor to the start of the next *word*.
W	Move the cursor to the start of the next *Word*.
b	Move the cursor to the start of the previous *word*.
B	Move the cursor to the start of the previous *Word*.
e	Move the cursor to the end of the next *word*.
E	Move the cursor to the end of the next *Word*.
f*x*	Move the cursor to the next instance of the character *x* in the current line.
F*x*	Move the cursor to the previous instance of the character *x* in the current line.
t*x*	Move the cursor forward to the character immediately before next instance of the character *x* in the current line.
T*x*	Move the cursor backward to the character immediately after next instance of the character *x* in the current line.
-	Move the cursor backward to the first character of the previous line.
+	Move the cursor forward to the first character of the next line.
}	Move forward to the next empty line following a non-empty line. This search stops at the end of the file.
{	Move backward to the previous empty line preceding a non-empty line. This search stops at the beginning of the file and does not wrap.
Ctrl-e	Add one more line at the bottom (end) of the screen (scroll down), but leaving the cursor on the same character.
Ctrl-y	Add one more line at the top of the screen (scroll up), but leaving the cursor on the same character.

Any sequence of keys that performs either a navigation or a search will now be identified as a *navsearch*.

5. Advanced editing

You should probably not memorize these. Instead, review them, and later when you begin editing larger files, you will likely find yourself thinking that the basic editing commands are too tedious or coarse, and you will then search for more efficient means of editing your file. Needing and then using a feature is a much better and memorable means of remembering that feature than just reading about them.

`o`	Open a new empty line after the line the cursor is currently on, move the cursor to that line and go into *insert mode*. (Same as `A`Enter)
`O`	Open a new empty line immediately before the line the cursor is currently on, move the cursor to that line and go into *insert mode*. (Same as `I`Enter`ki`)
`r`*x*	Replace the character on which the cursor is with the character *x*.
n`r`*x*	Repeat the next *n* characters starting at the cursor with the character *x*.
`R`	Go into *insert mode* but overwrite the text that is currently there.
n`R`	Repeat the inserted text *n* times.
`s` or *n*`s`	Delete the current character or *n* characters on the current line starting at the cursor and go into *insert mode*.
`S`	Delete the entire line and go into *insert mode*.
`~`	Switch the case of the letter on which the cursor is. Nothing happens to non-alphabetic characters.
n`~`	Switch the case of the next *n* letters starting at the cursor. Nothing happens to non-alphabetic characters.
`J`	Append (join) the next line onto the end of this line.
n`J`	Join next *n* lines together into one line starting with the line the cursor is currently on; however, `J`, `1J`, and `2J` all do the same thing.
`.`	Repeat the previous edit.

The default buffer

All of the following edits include an interaction with a *default buffer*. For example, in a Microsoft Word document, if you highlight text and then press either Ctrl-c or Ctrl-x, the highlighted text is placed into a buffer and in the latter case that text is also deleted from the Microsoft Word document. The text in that buffer can then be pasted into the document at another location using Ctrl-v.

There are two types of edits that have different interactions with how text is taken out of the buffer: character-to-character edits and line-to-line edits.

There are three different styles of editing text that deal with the buffer:

1. Deletions associated with the command `d` that delete the specified text and moves it to the buffer.
2. Copies (or **y**anks) associated with the command `y` that copies the specified text to the buffer.
3. Delete-and-insertions (or **c**hanges) associated with the command `c` that delete the specified text, moves it to the buffer, and then puts you into insert mode.

Visual highlighting

Another means of flagging text to be deleted, yanked or changed is to press `v` and then perform any sequence of navigation or searches. This continues to highlight text from the cursor to wherever the navigation or search would move the cursor. Similarly, if you press `V` and then perform any sequence of navigations or searches, it will highlight all lines from the current line to the line in which the cursor is moved as a result of the navigations or searches. We will indicate the pressing of `v` or `V` followed by an arbitrary number of navigations or searches by *visual*.

Advanced editing with the buffer

The following commands interact with the buffer as well as performing edits to the document:

`dd yy cc`	Delete, yank or change the entire current line.
`D C`	Delete or change the current line from the cursor to the end of the line.
`Y`	Yank the entire current line (thus, behaving more like `S` and not the previous two).
`d`*navsearch* `y`*navsearch* `c`*navsearch*	Delete, yank or change from the cursor to wherever the cursor would end up as a result of the search that is subsequently entered. `d}` would delete up to the end of the paragraph, `y/hi` would yank from the current cursor up to the start of the next instance of `hi`, and `cw` would delete up to the end of the current *word* and go into insert mode.
visual`d` *visual*`y` *visual*`c`	Delete, yank or change the text that are visually highlighted.
`diw yiw ciw` `diW yiW ciW`	Delete, yank or change the entire *word* or *Word* upon which the cursor is currently sitting.

Note that `s` and S put the characters or lines deleted, respectively, into the default buffer.

If the default buffer has been filled with the content of one of the above edits, the contents can be pasted (or put) using `p` or `P`:

1. If the buffer was filled from character-to-character, `p` will insert the content of the default buffer after the cursor, while `P` will insert the content of the default buffer before the cursor.
2. If the buffer was filled from line-to-line, `p` will insert the content of the default buffer after the current line containing the cursor, while `P` will insert the content of the default buffer before the current line.

Another edit is indenting (adding a tab at the start of the line) or unindenting (removing a tab from the start of the line) through `>` and `<`, respectively.

`>>` `<<`	Indent or unindent the current line.
`>`*navsearch* `<`*navsearch*	Indent or unindent all lines including the current line up to and including any line upon which the cursor falls as a result of the navigation or search.
visual`>` *visual*`<`	Indent or unindent all lines that have at least one character visually highlighted.

In C++ and similar languages, a common indentation is `>%`, which indents the current block.

Other commands that work with visual highlighting include:

visual`~`	Change the case of all letters that have been visually highlighted.
visual`J`	Join all lines that have at least one character highlighted.

There are many more such commands, but these are some of the most common this author uses.

6. Advanced search and search-and-replace

We will now describe more complex searching patterns, and other features we can

Regular expressions

Rather than just searching for explicit text, you can also search for patterns. For example, you may wish to find all instances of the word `Rational` or `rational`. One could search for `Rational` first, and then search for `rational`; however, that would be inefficient. A *regular expression* for both of these is

```
[Rr]ational
```

where the square brackets indicates that any letter within the brackets can be matched. This leads us to our next problem: how do we search for a bracket? To search for the bracket characters, these must be escaped (similar to how various characters must be escaped in C-style strings). The escape character is the same: the backslash, so to search for an explicit opening or closing square bracket, you must use `\[` and `\]`, and to search for a backslash, you must use `\\`.

The content of square brackets can be any set of characters you want; for example, if you wanted to match any vowel, you could use `[aeiouy]`. There are, however, short cuts by using a range of letters or numbers. For example, `[0-9a-fA-F]` matches a hexadecimal digit. Of course, this means that inside of bracketed characters, the dash must be escaped. Similarly, you can indicate that you want a pattern to not match any of the letters found by prefixing the sequence with a carat; for example, `[^aeiouy]` will match any character (including non-alphanumeric characters) that is not in the list. Consequently, again, inside the brackets, if you want to match a carat, you must escape it, and it might be a good idea to simply escape it regardless as to where it appears in the list.

There are some combinations of letters that are so frequently used that they have an escaped letter to represent them. First, the period `.` matches any character including whitespace, .

Digits

A decimal digit	`\d`	`[0-9]`	`\D`	`[^0-9]`
A hexadecimal digit	`\x`	`[0-9a-fA-F]`	`\X`	`[^0-9a-fA-F]`
An octal digit	`\o`	`[0-7]`	`\O`	`[^0-7]`

Letters of the alphabet

Letters	`\a`	`[a-zA-Z]`	`\A`	`[^a-zA-Z]`
Lower-case letters	`\l`	`[a-z]`	`\L`	`[^a-z]`
Upper-case letters	`\u`	`[A-Z]`	`\U`	`[^A-Z]`
Whitespace	`\s`	`[ \t]`	`\S`	`[^ \t]`

Variable names

You will recall that a variable name in C++ must start with a either a letter or an underscore, and all subsequent characters must be letters, the underscore or digits.

First character	`\h`	`[_a-zA-Z]`	`\H`	`[^_a-zA-Z]`
Subsequent characters	`\w`	`[_a-zA-Z0-9]`	`\W`	`[^_a-zA-Z0-9]`

Suppose you want to search for one of multiple options matching texts. Recall that in C++, `||` is a logical OR in C++. Similarly, you can match one of many regular expressions us an escaped pipe; for example, the following matches one of flow control keywords in C++:

```
for\|if\|do\|while
```

One issue with these matches is that it will match text like **for**tunate, T**if**fany and avaca**do**. You can specify that a regular expression must come at the start of a *name*[1] consisting of a contiguous sequence of letters, numbers or the underscore by prefixing the regular expression with `\<` and you can specify that the regular expression must end with a *name* by appending a `\>`.

```
\<for\>\|\<if\>\|\<do\>\|\<while\>
```

Alternatively, you can group regular expressions by wrapping them in escaped parentheses:

```
\<\(for\|if\|do\|while\)\>
```

You can use the escaped parentheses in a manner similar to parentheses in C++.

Suppose you want the regular expression to start at the beginning of a line or to appear at the end of a line. This can be specified by using `^` and `$`; for example, `^\s` matches a whitespace character at the start of a line, while `;$` matches a semicolon at the very end of a line. Consequently, you must also escape the dollar sign and the carat in usual regular expressions.

[1] Previously, *word* and *Word* were used for other combinations of characters, so here I use *name* as it describes a variable name but also allowing for that variable name to be prefixed by digits.

Finally, suppose you want to specify repetition. Suppose you want to match an `if` statement followed by zero or more whitespace characters followed by an opening parentheses. For this, we can use

```
\<if\s*(
```

to match the string `if` at the start of a name followed by zero or more whitespace characters followed by `(`. Because the star immediately follows the regular expression matching a single character, that single character will be matched zero or more times. You could also give a regular expression for describing a variable name in C++:

```
\<\h\w*\>  or  \<[_A-Za-z][_A-Za-z0-9]*\>
```

If you wanted to match zero or more repetitions of a more complex regular expression, simply wrap that expression in escaped parentheses; for example, you could match Eduard Khil's song as

```
\<[Tt]ro\(lo\)*\>
```

which will match the words `Tro`, `tro`, `Trolo`, `trololololo`, etc. If you wanted to ensure at least one "lo" at the end, you could explicitly add it it:

```
\<[Tt]rolo\(lo\)*\>
```

However, instead of matching zero or more of a given regular expression, there are six other means of matching various numbers of matches of a given regular expression, including:

*regex**	Match zero or more instances
regex\+	Match one or more instances
regex\=	Match zero or one instances
regex\{*m*}	Match exactly *m* instances
regex\{*m*,*n*}	Match between *m* and *n* instances
regex\{m,}	Match *m* or more instances
regex\{,n}	Match zero to *n* instances

For example, a regular expression that identifies a declaration of an `int` on a line by itself is:

```
^\s*int \h\w*\({[+-]\=\(\d\+\|0x\x\)}\)\=;$
```

which describes:

1. the start of a line `^`,
2. followed by zero or more whitespace characters (spaces or tabs) `\s*`,
3. followed by `int` and a space,
4. followed by a variable name `\h\w*`,
5. optionally followed by either zero or one of `\({\s*[+-]\=\s*\(\d\+\|0x\x\+\)\s*}\s*\)\=` which consists of
    a. an opening brace `{`,
    b. followed by zero or more whitespace characters `\s\*`,
    c. followed by either of `\(\d\+\|0x\x\+\)` which are
        i. a sequence of decimal digits
    d. an opening brace `{`,
    e. an opening brace `{`,
    f. an opening brace `{`,
    g.
6. followed by a semicolon `;`,
7. followed by zero or more whitespace characters `\s\*`,
8. followed by the end of the line `$`.

This would match any of the following declarations:

```
   int s{32};
int my_var2{-234};
           int an_odd_variable_name_70{+616};
```

It would not, however, match any of the following:

```
   int s{32}, t{17}, u{19};
int my_var2{ -234 };
           int   an_odd_variable_name_70 { +616  }   ;
```

all of which are valid declarations. How would you modify the above regular expression to match these, as well?

Back-references

When doing a substitution, it may be useful to refer back to the text that originally matched expression. For example, suppose you have read the scientific literature and understand that `CamelCase` for variable names is less effective than `Snake_case`, so you would like to do a global substitution of all Java-naming-conventions to C-style naming convetions. After some thought, you realize all you need to do is replace all instances of a lower-case letter followed by an upper-case letter with the lower-case letter, an underscore and the upper-case letter switched to lower case. The pattern you are looking for is `\l\u`, but what now? Any matched text inside escaped parentheses can be referred to in the replacement text; specifically, all text that matches the first opening `\(` can be referred to by `\1`, all text that matches the second opening `\(` can be referred to by `\2`, and so on, up to `\9`. We must refer to the lower-case and upper-case letters separately, so we could start with

```
:%s/\(\l\)\(\u\)/\1_\2/g
```

This would change `CamelCase` to `Camel_Case`. Now we would also modify the second match: the following commands make changes to the substituted text:

`\r`	Introduce a new line (carriage return).
`\l`	Make the next character matched lower case if it is a letter.
`\u`	Make the next character matched upper case if it is a letter.
`\L…\E`	Make all letters up to `\E` lower case.
`\U…\E`	Make all letters up to `\E` upper case.
	Match *m* or more instances
	Match zero to *n* instances

Thus, we could use the expression

```
:%s/\(\l\)\(\u\)/\1_\l\2/g
```

And the `\l` will change the first character of `\2` (which happens to be only one character long) to lower case.

7. Named buffers

To this point, there is only the default buffer, and any yank, delete or change copies information to that buffer, and any put operation copies that buffer back into the file. There are, however, an addition 52 named buffers, each one identified by either a lower case or upper case letter. Any time you are about to perform either a yank, delete, change or put, if you want to instead copy to or paste from a named buffer, you prefix the `y`, `Y`, `d`, `D`, `c`, `C`, `p` or `P` with `"`***`a`*** where ***a*** is any letter from `a` to `z` or from `A` to `Z`.

8. Bookmarks (markers)

It is also possible to insert bookmarks (or *markers*) throughout your file. Each lower and upper case letter may be used to represent a different bookmark and you can use these bookmarks as follows:

`m`***a***	Add (or *m*ark) a bookmark ***a*** to the line the cursor is current on where ***a*** is any letter of the alphabet, either upper case or lower case.
`'`***a***	Jump to the first non-white-space character of the line marked by ***a***.

If a line with a bookmark is deleted, the bookmark is removed. If a line with a bookmark is joined with another line, the bookmark becomes associated with the joined line. Bookmarks cannot be cut and pasted elsewhere in the document. If a line with a bookmark is cut and pasted elsewhere, the bookmark must be set again.

9. Editing multiple files

Suppose you need to edit multiple files. At the command line, you may issue a command like

```
vi *.cpp
```

if you were attempting to edit multiple source files simultaneously, or perhaps you are editing a number of web pages. *vi* allows you to edit one document at a time, but you can go between the documents by using the following escape commands:

`:n` `:N`	Go to the next or previous file to be edited.
`:n!` `:N!`	Go to the next or previous file without saving the current one (if it is not saved).
`:wn` `:wN`	Save (write) the current file and go to the next or previous file to be edited.

Each of these can be modified by prefixing an integer indicating the number of files to go either forwards or backwards; for example, `:2wn` writes the current file and then goes forward two documents.

Additional comments by the author

Most students who have been previously exposed to *vi* may wonder why I use Ctrl-[and not Esc. The answer is simple: if you can touch type and you use Esc, then Esc is the only key that forces your hand to leave the home row.

Some other readers may say that there are different techniques of doing similar navigations or edits that have not been described here. The answer here is that rather than overloading the reader with two or three ways of doing the same operation, generally one technique is provided, and that technique generally follows the same pattern of other commands, so that the reader may be able to intuitively understand the various commands. The one command this author does not understand is why `D` and `C` perform their operation from the cursor to the end of the line, while `Y` yanks the entire line in a manner closer to `S`.

My favorite escape command is `:set ts=3`Enter, which has each Tab character (a tab stop) displayed as three spaces and not the default eight spaces.

Here is a state diagram of how *vi* works: