Vim, a Popular Text Editore

Bram Moolenaar

Updated on Dec 20, 2001

Abstract

The main author of the free text editor Vim writes about how it came to be and how it is being developed. The Charityware concept is discussed and why Bram chose to use it for Vim. Diving into the data structures and functions to manipulate them, he gives you some idea of how this complex program works. Some of the new features that Bram has added in Vim 6.0 are discussed.

1 Introduction

There is a small chance you have never heard about Vim. It is included with almost every GNU/Linux distribution as the standard Vi editor. For many systems such as FreeBSD and Sun Solaris 8 it comes as an extra package you can easily install. For other systems (MS-Windows, Macintosh, OS/2, etc.) executables and source code can be downloaded from many places on the internet.

Vim is a text editor like Vi, but with many extra features; of the Vi-like editors Vim has often been voted the best — in fact the main competition comes from Emacs, in all its different versions. In 1999 Emacs was selected the winner in the Linuxworld text editing category, with Vim second, but in February 2000 Vim won the Slashdot Beanie award for best open source text editor pushing Emacs into second place. So these editors are in the same league.

One of the main advantages of Vim (and Vi) is that the most often used commands are typed with alphabetic characters. This is much faster than using Control or Meta key combinations, and less stressful on the fingers. The extra time it takes to learn the commands is soon repaid by allowing you to work efficiently. However Vim, unlike Vi, also supports the use of cursor keys in Insert mode and so a beginner can get started quickly before gradually learning more commands to gain speed.

Compared to other Vi-like editors Vim has very many features: syntax highlighting for over 180 languages, automatic indenting for C code, a powerful script language, etc. There is something for everyone.

The development of Vim is ongoing. At the moment of writing, the work for Vim version 6.0 has finished in September; more about that further on. The main goal for Vim is to be a very stable and useful tool; besides adding more features, improving the existing features and fixing bugs is pursued very actively. Compare that to the original Vi, for which not much work has been done since 1985.

This picture shows the GTK version of Vim, editing C source code with syntax highlighting. The search pattern `"msg_didany"` is highlighted with a yellow background. There is a Visual selection (grey background) with a green, blinking cursor. At the bottom `"-- VISUAL--"` indicates the current mode and at the right the position in the file is shown.

2 History of Vim

A long time ago I got myself an Amiga computer. Since I was used to editing with Vi, I looked around for a program like Vi for the Amiga. I did find a few so-called 'clones', but none of them was good enough; so I took the best one, and started improving it. At first the main goal was to be able to do all that Vi could do. Gradually I added some additional features, like multi-level undo.

When it was working reasonably well, I released a version of Vim (then called "Vi IMitation") on a public domain disk set for the Amiga, made by Fred Fish. Then others started sending me patches. A few people took the effort to port Vim to other platforms, like MS-DOS and Unix. I added more features and made it work better. By that time it was justified to rename it to "Vi IMproved". Over time the code has been redesigned and extended so much that almost nothing of the original 'clone' remains.

When I started working on Vim it was just for my own use. After some time I got the impression it was useful for others, and sent it out into the world. Since then I'm working more and more on making the program work well for a large audience. It's fun to create something useful. Also, there is a nice group of co-authors and power users, which is very inspiring.

Here is an overview of Vim's history:

```
1991 Nov 2  - Vim 1.14:
First release (on Fred Fish disk #591).

1992        - Vim 1.22:
Port to Unix.  Vim now competes with Vi.

1994 Aug 12 - Vim 3.0:
Support for multiple buffers and windows.

1996 May 29 - Vim 4.0:
Graphical User Interface (largely by
Robert Webb).

1998 Feb 19 - Vim 5.0:
Syntax coloring/highlighting.

2001 Sep 26 - Vim 6.0:
folding, plugins, vertical split
```

2.1 What does IMproved stand for?

By version 1.22 Vim included more features than Vi. I decided to change the name from "Vi IMitation" to "Vi IMproved". Over time the gap has grown wider, now it's hard to think of a reason to use Vi instead of Vim. I would recommend all Vi users to switch to Vim. The list of advantages is very long, here are just a few:

```
- Unlimited line length and allow NUL bytes:
  Possibility to edit any file, including
  binary files.
- Multi-level undo and redo: Don't worry
  about destroying your file when caps-lock
  was accidentally on.
- Multiple windows and buffers: Edit
  several files at the same time, copy
  text between them.
- Syntax highlighting: Quickly understand
  the structure of the text and spot errors.
- Command line history and completion:
  Correct typos, recall old commands,
  quickly enter long file names.
- Delete and put rectangular text areas:
  Edit a table.
- Error message parsing: Run the compiler
  and immediately jump to locations where
  errors have been found.
- On-line help with hyperlinks: Find
  comprehensive documentation for any
  command and jump to related subjects.
- A powerful script language: Add your
  own extensions.
```

2.2 Developing Vim

After a couple of years of hard work version 6.0 is now finished. I have done the core items myself, but many people join in. Sometimes they create a new feature and send me a patch that I can include right away, but mostly the patches I receive need to be integrated into the current version. Few people can overview how all the parts of Vim work together, since the code has become quite complex. For example, someone created a patch for multi-line search patterns. This showed how it could be done, and the locations in the sources that had to be changed. Unfortunately the author used pointers to text lines, and didn't realise that these can become invalid. I had to go through all the new code to fix this. Although that was a lot of work, the patch I got helped me by setting me on the right track.

Users ask questions on the Vim mailing lists, which indicates to me what the most common problems are. Sometimes people send me patches or requests for new features. I think that this co-operation of users and co-developers is the main strength of how Vim is being developed. Users and developers communicate directly with each other and with an open spirit. This is how open source software can grow to become better than commercial software.

3 Distribution

Vim can be distributed freely; however, there are some modest restrictions. This is the text that comes with Vim:

Vim is Charityware. You can use and copy it as much as you like, but you are encouraged to make a donation to orphans in Uganda. See `iccf|` below.

If you include Vim on a CD-ROM, I would like to receive a copy. Just so I know which Vim distributions exists in the world (and to show off to my friends :-)).

DETAILS

There are no restrictions on distributing an unmodified copy of Vim. Parts of Vim may also be distributed, but this text must always be included. You are allowed to include executables that you made from the unmodified Vim sources, your own usage examples and Vim scripts.

If you distribute a modified version of Vim, you are encouraged to send the maintainer a copy, including the source code. Or make it available to the maintainer through ftp; let him know where it can be found. If the number of changes is small (e.g., a modified Makefile) e-mailing the diffs will do. When the maintainer asks for it (in any way) you must make your changes, including source code, available to him.

The maintainer reserves the right to include any changes in the official version of Vim. This is negotiable. You are not allowed to distribute a modified version of Vim when you are not willing to make the source code available to the maintainer.

The current maintainer is Bram Moolenaar . If this changes, it will be announced in appropriate places (most likely www.vim.org and comp.editors). When it is completely impossible to contact the maintainer, the obligation to send him modified source code is dropped.

It is not allowed to remove these restrictions from the distribution of the Vim sources or parts of it. These restrictions may also be used for previous Vim releases instead of the text that was included with it.
`SUMMARY`

```
Vim is Charityware.  You can use and
copy it as much as you like, but you are
encouraged to make a donation to orphans in
Uganda.  See iccf| below.

If you include Vim on a CD-ROM, I would
like to receive a copy.  Just so I know
```

```
which Vim distributions exists in the world
(and to show off to my friends :-)).

DETAILS

There are no restrictions on distributing
an unmodified copy of Vim.  Parts of Vim
may also be distributed, but this text
must always be included.  You are allowed
to include executables that you made from
the unmodified Vim sources, your own usage
examples and Vim scripts.

If you distribute a modified version
of Vim, you are encouraged to send the
maintainer a copy, including the source
code.  Or make it available to the
maintainer through ftp; let him know where
it can be found.  If the number of changes
is small (e.g., a modified Makefile)
e-mailing the diffs will do.  When the
maintainer asks for it (in any way) you
must make your changes, including source
code, available to him.

The maintainer reserves the right to
include any changes in the official version
of Vim.  This is negotiable.  You are not
allowed to distribute a modified version
of Vim when you are not willing to make the
source code available to the maintainer.

The current maintainer is Bram Moolenaar
.  If this changes, it will be announced in
appropriate places (most likely www.vim.org
and comp.editors).  When it is completely
impossible to contact the maintainer, the
obligation to send him modified source code
is dropped.

It is not allowed to remove these
restrictions from the distribution of
the Vim sources or parts of it.  These
restrictions may also be used for previous
Vim releases instead of the text that was
included with it.
```

I prefer to give users much freedom in using the Vim source code. The main reason to add restrictions is to avoid what happened to Elvis some time ago: someone took the Elvis source code, added a few nice Windows GUI things, and started selling it. Since those changes were not available as source code and most of that editor was still the original Elvis code, that didn't sound fair. Not only because people have to pay one guy for software that someone else made, but also because the author refused to publish the modified source code and allow others to further improve it. That's why I added the restriction that the source code of modifications must be made available to me. That still leaves room for a company to make a modified version of Vim and negotiate with me if their changes must be made public or not. This gives me the right to decide what happens with the software I created.

3.1 The future of open-source software

Which software will be open-source and which will not? I don't think there is a definitive answer to this question. For a given application, is there someone willing to create and maintain it as open-source software? That depends a lot on the motivation someone has to spend time on this without being paid for it. That is an uncontrollable process, with unpredictable results. Not too long ago it was thought that only small programs would be open-source, since large programs would require the long-term commitment and large investment that only commercial companies could afford. The development of Linux has proven this to be wrong, and that's not a one-off exception. Several other large projects have popped up and are successful, such as KDE.

In practice I see that most people are only motivated to make software they would use themselves. That's certainly the case for Vim: I use it myself every day. This is hardly a restriction though, since more and more people are learning how to write programs. It does restrict the number of people that are motivated to spend time working on a specific program. Theo-

retically it would be possible to compute the number of people that could work an open-source program:

```
available = interested x ability x motivation
```

where:

available — number of people available to work on an open-source program

interested — number of people interested in using the program

ability — percentage of this group that are able to write the program

motivation — chance that someone is motivated to write the program

Note that the number of people interested in the program also depends on the availability of existing programs. If there is no software that is good enough or it is too expensive, this number will be higher. If there's already a program available that's cheap and good, the number will be much lower.

Not everybody is able to write software and there is a large difference in required skills for different programs. If the goal is to make a program for software engineering, there is a large chance that the target audience includes people that have the skills to make the software. If the program is to process data for rare birds, the percentage is much smaller.

The motivation factor is the big unknown in this formula. How many of the people that are able to write the program are actually willing to do it? It would be interesting to perform a study on this, and find out if this factor can actually be estimated.

A correction to the formula needs to be made for people who are willing to write a program for others. That applies especially to software written for the disabled. Otherwise, I think this group is quite small.

If we have computed the number of people who are available to write the program, the big question is if it will actually happen. Or better: When it will happen. Given enough time, I'm sure that every program for which there is a need will be made. It should be possible to make a formula to compute the chance that the program will be written this year. That's left as an exercise to the reader ...

Overall there are a lot of undetermined factors in this formula. I would conclude that it's unpredictable how much open-source software there will be in the future.

4 Charityware

Since Vim is open-source and freely distributable, users don't have to pay to use it. Even so quite a few people who use Vim regularly expressed to me that they wanted to reward me for my work in some way. I didn't really need extra money myself and didn't like the idea of some people giving me money for a program that is free. That's when I thought of the Charityware concept. The basic idea is that everybody who uses Vim is asked to donate to a charity. Thus the use of Vim is free, but if you think it's worth something, give that money to a good cause.

How did I chose the charity? Well, I have worked for a year as a volunteer with a project in the south of Uganda. This is an area that has been struck hard by AIDS. Estimates are that 10 to 30adults are infected by HIV. Many parents die, leaving their children behind. The project helps these needy children in several ways. We find a new home for the child. We make sure the child can go to school, gets medical attention and care made to measure.

After I returned from Uganda, my heart was still there. I decided the least I could do was to keep supporting the project by raising money for them. The connection to Vim was a very logical one. Thus now I'm asking Vim users to consider donating for the orphans in Uganda. I have also setup an adoption program. You can financially adopt a child, which

means that the child gets long-lasting help, which is best for the child. Since we work only with volunteers and the money is directly sent to the project, almost all the money is really used in Uganda.

You can find more information about this at *http://iccf-holland.org.*

(Nabasagi Morine is one of the children sponsored through ICCF Holland.)

4.1 Does the Charityware concept really work?

I have received many donations for the orphans in Uganda through the Charityware concept. Very irregular, sometimes nothing for a month and then several at once. Some small amounts, some quite big. I can't mention the exact amounts, because not all donations go through me, and it is not always clear if a donation was done because of Vim. An estimate is that we received $2000 in 1997 and $4000 in 1998. A remarkable portion comes from Germany.

It is not only the money. The Charityware concept also helps to make people aware of the need of other people. If I would not have started this, few people would know about the project in Uganda. For many people it takes time to get used to the idea that there is more to life than making money and taking care of yourself. I have had reactions from people who could not afford to donate, but were moved by the concept. In some way it seems Charityware changes people a bit.

4.2 Is Charityware a good idea for other programs?

Check these arguments whether it is a suited to your program:

- You don't need the money yourself. If you like spending money on a bigger car than your neighbour, Charityware is probably not for you.
- You currently provide your program as shareware, but you don't get much money from it, because people don't like giving it to you.
- You are currently providing the program completely for free, and think it is worth something.

When you decide to go for Charityware, what good cause can you use? The best is one with which you are in contact personally. Users will then better understand your motivation. Otherwise, find a small organisation that you trust and needs more money and attention. I discourage supporting large organisations, especially if they already have enough ways to contact potential sponsors.

4.3 Variations on Charityware

I do not force Vim users to make a donation to the project in Uganda. And there is no amount specified. This gives a lot of freedom to the user. He may read the note about AIDS orphans and then forget about it.

Alternatively, users can be told that they **_must_** make a donation, and specify an amount. This remark can be put in a place that does not go unnoticed for the user. Hopefully this will generate more donations, but be careful not to annoy the user. You must set a minimum, otherwise the enforcement to donate is meaningless. It will be difficult to decide on an amount though.

A more forceful approach would be to enforce the payment by blocking some or all functionality until the donation has been made. This is not a good option since it conflicts with the charitable ethic; poor people would not be able to use the program.

5 Vim internals

Vim has grown into a very large program. Here is an overview of the main parts and in which source files they can be found. I have included the file size for Vim 5.6, to give you an idea of how much code is required for each part.

```
startup code, main command loop
main.c       43538

handling Normal mode commands
normal.c     140320

executing operator commands
ops.c        109742

displaying text on the screen
screen.c     165136

multi-window handling
window.c     52168

multi-buffer handling
buffer.c     63743

typeahead, mappings and abbreviations
getchar.c    76964

terminal input and output
term.c       109869

reading and writing files
fileio.c     122685

swap file management
memfile.c    31516

managing lines of buffers
memline.c    114193

undo and redo
undo.c       29014

Insert mode
edit.c       142302

keeping marks in files
mark.c       25582

editing the command line
ex_getln.c   89509

executing Ex commands
ex_docmd.c   176724

complex Ex commands
ex_cmds.c    100549

Quickfix commands
quickfix.c   31760

regular expression matching
regexp.c     73771

search commands
search.c     93721

tag commands
tag.c        69337

setting options
option.c     148884

Vim script commmands, expression evaluation
eval.c       122346

System-specific code
os_*.c up to 94640

Generic GUI code
gui.c        73468

Motif GUI code
gui_motif.c  26963

Athena GUI code
gui_athena.c 25220
```

```
common Motif and Athena GUI code
gui_x11.c    62913

GTK GUI code
gui_gtk*.c   158289

MS-Windows GUI code
gui_w32.c    141809

Macintosh GUI code
gui_mac.c    82600

menu handling
menu.c       36164

Perl interface
if_perl.c    21246

Python interface
if_python.c  51814

Sniff interface
if_sniff.c   25035
```

Notice that the files are generally big; this is because I prefer to keep related items together in one file. There could be more and shorter files. That would be preferred when using a revision control system. For me the current way of working doesn't cause problems, and I don't see a good reason to spend time on changing it. All files are together in one directory. The main advantage is that a simple "grep" command can be used to find all locations where an identifier is used.

There are a few remarkable items in the list. The undo functionality is relatively small. That's because it uses a well thought-out mechanism, which is simple and powerful. On the other hand, you can see that the GUI files are generally big. Coding for the GUI just tends to be a lot of work.

Professor Michael Godfrey has done some studies on the Vim architecture and its development. You can find this at *http://plg.uwaterloo.ca/~migod/*.

Following are a few specific items that you might find interesting.

5.1 Portability

Vim was made such that it works on many different operating systems. This wasn't easy. Just supporting the most popular Unix versions is already a big task. Adding support for MS-DOS and MS-Windows gives the additional problem of using file names with a backslash in them. Machines such as the Amiga and the Macintosh have a different kind of operating system, which require specific solutions.

One of the main choices in Vim was to use the good old K&R C. This can be compiled on just about any system. The advantages of using ANSI C can be mixed into this, by using the pre-processor. For example, function prototypes look like this:

```
void ml_open_file __ARGS((BUF *buf));
```

For non-ANSI compilers the `"__ARGS(())"` part evaluates to `"()"`. These prototypes are generated with the cproto program. This not only avoids the need to type these prototypes, it also avoids errors. It's a bit more work to use the `__ARGS()` construct, but it makes Vim work on old systems, and still allows modern compilers to check the function prototypes, which avoids a lot of trouble.

To be able to compile Vim on many different systems, there is a system-specific file for each of them, such as `"os_unix.c"` and `"os_amiga.c"`. It contains most of the system-dependent code. However, this doesn't cover everything. There are a lot of `"#ifdef"` constructs in the generic files. Sometimes I clean this up to avoid the clutter. This involves creating a `"mch_xxx()"` function in each of the system-specific files, and calling it from the generic files. Since I don't have all these different systems, it requires co-developers to test this change.

For Unix "autoconf" is used to detect all kinds of things about the system. It finds out which include files there are, the location of libraries, etc. This uses a description file with the tests to be done, called `"configure.in"`. Autoconf processes this file and produces the `"configure"` shell script. Care has been taken that this shell script can be run on any Unix system. Although using autoconf isn't easy, I still recommend it. If you want your program to be portable over different Unix systems, autoconf is the way to go.

The standard way is for the user to run `"configure"`, which generates the Makefile. Vim is a bit different, because there is already a Makefile in the distribution. When running "make" for the first time, it will start configure for you, with the default arguments. This generates a `"config.mk"` file, which is included from the Makefile. The main advantage of this different approach is that the Makefile can contain examples of configure arguments, which you can uncomment. Otherwise you would have to figure out the configure arguments by running `"configure --help"`, inspecting its output and typing the arguments to configure. For Vim you can edit the Makefile, read the comments, and remove or insert a few `"#"` characters right there. This is much more convenient. Moreover, running configure directly still works, thus this approach doesn't have drawbacks.

5.2 Storing text

The main task of a text editor is to read a file, modify it and write it. There are quite a few demands for how the text is kept inside the editor:

```
- Reading and writing files should be fast.
- No restrictions on line length, file
  length or character set.
- Making changes in the text should be
  quick.
- The code to handle the text shouldn't be
  too big.
- Allow for undoing changes.
- When the system crashes, recovering
  unsaved changes must be possible.
```

There is no easy way to meet all these. First of all, it's not possible to keep all text in memory. Some systems just don't have enough, and for others it's very inefficient to use a lot of memory. That is why I chose to use a swap file. File access can be slow, so some of the text should be kept in memory via a caching mechanism. Since file I/O works faster when using fixed-size blocks of a few Kbyte, it's better not to write individual text lines to the swap file. After considering a few alternatives, I decided to use a construct that stores a number of lines in a block. These blocks are handled similar to the BSD `"db"` functions, but optimised for how Vim uses text.

You can find the code to deal with blocks of text lines in memfile.c. It maintains a cache to reduce the disk I/O. This works very much like any cache system: blocks are written to the swap file when using too much memory, or to save changes that the user made. This allows recovering the changes from the swap file, in the event that the system goes down unexpectedly

Packing text lines in a block is done in memline.c. Since the blocks are fixed in size, the number of lines that fit in them varies. Quite a lot of code is required to handle this, to be able to insert lines, delete lines and change lines. The complexity mostly comes from situations where an inserted line doesn't fit in an existing block, requiring the need to split that block in two. There is also an exception: If there is a single line that doesn't fit in a normal block, a block is created that is big enough to contain the line. This allows handling lines of any length, without compromising the efficiency of handling for short lines.

The blocks with text lines are stored in the swap file without a specific ordering. If the blocks were ordered, inserting a block halfway into the file would require all remaining blocks to be shifted, which is very slow. To be able to find a line by its number,

index blocks are used. An index block contains a list that tells which line is in which block. If a file is big, this list doesn't fit in a single block. It is then split over several blocks, and another index block is made to refer to these index blocks. This forms a balanced tree of index blocks, with the text blocks as the leaves. This construction has proven to be very reliable and efficient.

5.3 Syntax highlighting

Parsing a file to recognize its structure can be complex. Add to that the demand that parsing might have to start anywhere in the file, and you end up with a difficult task. Fortunately we were able to split the task into two pieces: The core syntax engine, which has been implemented in C, and the specification of the syntax items for each language with patterns. This allows adding support for a new syntax without recompiling Vim.

The basic idea is that a regular expression pattern is used to specify an item of the language. Vim tries to match that pattern in each line of the text. If it matches, the matching text is highlighted. For items that are kept within one line this is quite simple to implement. For items that cross a line boundary it becomes more complicated. In Vim you need to specify a pattern for where the item starts and where it ends. For example, a C comment starts with `"/*"` and ends with `"*/"`. But what happens when Vim starts displaying a line halfway a comment? There is no `"/*"` there, thus it would not be recognized as a comment line. This requires searching backwards in the text to find out if the text is inside a comment or not. This is called syncing.

Another construct is nesting. This happens if you want to highlight the word `"TODO"` when it's inside a comment. You then have a todo syntax item inside a comment syntax item. On the other hand, if you find a "/*" inside a string, it is not the beginning of a comment. Thus a string item cannot contain a comment item. All this is done with contained items. For each syntax item the user can specify which other syntax items it may contain. This fits quite well with the structure of most languages.

The syntax code maintains a stack of nested syntax items. For any point in the file, Vim will look for items that can be contained in the item that is currently at the top of the stack. Thus this stack represents the current state of the syntax engine. To speed up the syntax highlighting, this state is saved and used again later when the same line is displayed. When making changes to the text the saved syntax state may become invalid. A change in one line may cause a change in syntax for many following lines. For example: when inserting `"/*"` to comment-out some code, all syntax states until the matching `"*/"` will become invalid. The code notices that the state after the "*/" is the same as the old state, and so all states after this will also be valid. Thus recomputing the syntax can stop at this point.

5.4 Data structures

Vim uses many different data structures to store information. There is a trade-off between using a generic kind of structure and using a structure specifically tailored to an application. For example, a string can be stored in a fixed size array on the stack, a fixed sized chunk of allocated memory or a dynamically sized chunk of allocated memory. Which one to use depends on the type of string to be stored:

1. For strings that have a known maximum size, using the stack has the advantage that the time consuming malloc() and free() don't have to be used. This can usually be done with file names, but Vim tries not to restrict the user, thus other strings can be of any length and the stack is not suitable.

2. A fixed size of allocated memory can be used for items that don't change often. This is used for option values, for example. It doesn't waste

memory and the overhead of malloc()/free() is not important.

3. For strings that change often, calling malloc()/free() often consumes a lot of time. This is especially true for strings and lists that can grow in size. In Vim a dynamically allocated piece of memory is used for this - a "struct growarray". The memory allocation is made in large steps, so that a certain number of items can be appended without reallocating.

I have ended up creating quite a few data structures in Vim. The disadvantage is that this has taken a lot of time, it adds to the complexity and makes maintenance more work. The advantage is that Vim works very fast and efficient. Compare this with Emacs, where the main complaint is that it's slow and consumes a lot of resources. I have the impression that Emacs uses generic constructs and depends on the computer to be fast. In my opinion I have made the right choice. A text editor is used all day by many people; it is worth investing time and effort in making it work well.

6 The future

In November 1998 an survey was held to allow Vim users to vote for changes to Vim. This resulted in a good overview of what users wanted to be added. This is the top six:

1. `Add folding (display only a selected part of the text)`
2. `Vertically split windows (side-by-side)`
3. `Add configurable auto-indenting for many languages (like 'cindent')`
4. `Fix all problems, big and small; make Vim more robust`
5. `Add Perl compatible search pattern`
6. `Search patterns that cross line boundaries`

The first three and the sixth items have been implemented in Vim 6.0. Fixing problems continues as usual! Search patterns have been extended to include all the Perl features, but they are not directly Perl compatible. That would cause problems with backwards compatibility.

6.1 Folding

One of the first features that was added to Vim 6.0 is folding. This is a mechanism to hide part of the text, so that the overall structure can be seen. It looks like folding paper. This is a big change, impacting on all parts of the code - and it's not very clear how a user would use folding for different kinds of files. I have already spent quite a bit of time trying to come up with a good user interface for folding. Since different users have different demands, I ended up with several folding modes:

manual — The range of lines that forms a fold is selected manually.

indent — The indent of a line is used to decide if it will be folded. For example, all lines which have an indent of more than 8 spaces can be folded.

expr — Like "indent", but use an expression to decide about the fold level of a line.

syntax — The syntax highlighting items specify fold regions.

marker — Markers in the text are used to specify the start and end of fold region.

Most methods allow folding without changing the file. That is required for most people who are just viewing a file, or are sharing files with other people. But it doesn't allow specifying exactly where the folds need to be. Therefore the "marker" method has been added. This allows you to insert markers exactly where a fold should start. For example, if you have a few comments just before a function definition, y-

ou can put the marker just above the comment that should be in the same fold as the function. You can also include some text to explain the contents of the fold, for example "local declarations" or the name of a function and a short note about it. The disadvantage is that the file needs to be modified to add the markers.

This picture shows C source code with folds. The 'foldlevel' is set to show one fold line for each function. These folds have been defined with markers in the text, like "1", with a short text to explain the contents of the fold. The fold for the "lineFolded()" function has been opened to be able to view it.

6.2 Multi-line search patterns

As mentioned above, I have added the ability to match a line-break with a regular expression pattern. The changes to the documentation for this new feature are quite short; it's easy to explain the new patterns which can be used to match a line-break. The changes to the source code were more extensive. The main issue is that only one line of text from a buffer is guaranteed to remain in memory. When getting another line, the previous one may be freed to make room. This limits the amount of memory used. The code that was calling the pattern matching function often assumed that its pointers would remain valid, but now that this function may need to retrieve other lines in the buffer to check for a match, this is no longer the case. The solution is to use locations in the file - line number and column - instead of pointers

I thought that including the patch for multi-line patterns would be a day of work, but it turned out a week's worth of changes. It's easy to underestimate the impact of a change. The true scale often becomes clear only during the implementation and testing of the new functionality. If this happens in commercial software development you miss your deadline, go over the budget and are generally in trouble; thus it is necessary to carefully inspect the impact of a change and spend time on making a plan. Fortunately, for free software development this isn't all that important. You just end up spending some more time and the release is ready a bit later. Therefore it is often not worth incurring the overhead of planning. That's a big advantage of free software development over commercial software!

6.3 UTF-8

Another item that has made it into version 6.0 is internationalisation (i18n). Vim already supports editing double-byte files. This is called multi-byte support, but it actually doesn't work for more than two bytes. Since most of the world appears to be moving to Unicode, that is what I have added. Historically Vim has internally worked with bytes; therefore it's a logical step to use UTF-8 encoding (Unicode characters encoded as a sequence of 8-bit bytes). An alternative would be to use wide characters (16 or 32 bit), but that would require all code that handles text to be changed. Since most text would still be 7-bit ASCII, it would also be inefficient.

When adding support for UTF-8 there are a few important decisions to be made. Will all text inside Vim be encoded with UTF-8, or is there an option that switches between ASCII, UTF-8 and perhaps double-byte encoding? Doing everything in UTF-8 has the advantage that most of the code will be simpler, since it only has to deal with one type of encoding. But typing characters, displaying text and copy/paste with other applications may use another encoding, requiring conversion between UTF-8 and other formats. This could cause overhead and even changes in the text. The latter is anacceptable and must be avoided. If you read a file with Vim and write it without making changes, you should get exactly the same file.

Thus the main question is whether the text should be converted to UTF-8 when it is read into Vim, or kept in its original encoding:

Keep all text internally in UTF-8 format.

When reading a file, convert it to UTF-8, when writing it convert it to the file format. Typed text also needs to be converted, if the input method does not supply UTF-8 format. If the screen uses another format, it will also require conversion.

Keep the text in the original format of the file.

The typed text and characters sent to the screen only need to be converted when they use another format. All functions that handle text must be aware of the possible formats. Conversion needs to be done when yanking text in one window and putting it in another if the files use a different encoding.

Both alternatives have their advantages and disadvantages. After thinking about this for a while, I decided to implement a mix of 'both. The internal format can be selected. It can be ASCII, UTF-8, a double-byte encoding or something else. Usually this should match the environment (the current locale). When possible, conversion is done between the file format and the internal format. When the conversion is not possible (it is unavailable or there are illegal characters) it is skipped This possibly results in wrongly displayed text, but at least the text isn't modified when it is written back.

7 Conclusion

Vim has grown to become a big open-source program. Not only is it useful for many people, it can also be used as an example of how open-source programs are developed. I hope Vim inspires and helps other open-source authors with their work.

If you want to use Vim, you can find it in many places: if you have a GNU/Linux, FreeBSD or Solaris 8 distribution, look for a Vim package which you can install easily. For Linux the Vi command may start a minimal version of Vim, install a package if you want to use more features.

For information about downloading Vim use this *URL: http://vim.sf.net/download.php.*

A lot of information about Vim can be found on its web site: *http://www.vim.org*. Fortunately Sven Guckes volunteered to maintain the Vim web site, allowing me to concentrate on developing Vim. Vim tips and plugins can be found on *http://vim.sf.net*. These are created and uploaded by Vim users.

If you have questions, bug reports, or want to help with Vim development, join one of the Vim mailing lists. See *http://www.vim.org/mail.html*.

About the Author Bram Moolenaar studied electrical engineering at the Technical University of Delft and graduated in 1985 on a multi-processor Unix architecture. Although now mainly working on software, he still knows how to use a soldering iron. He was born in the Netherlands, in the area where tulips are grown, and now lives in Venlo, in the east of that country. He worked many years for Oce to design their first digital photocopying machines; he is currently doing freelance work and spends a lot of time on the free, open-source text editor Vim (*http://www.vim.org*), of which he is the main author. This involves handling e-mail from users and co-developers, fixing bugs and implementing new features. Bram founded the ICCF Holland foundation (*http://iccf-holland.org*) to support the Kibaale Children's Centre in Uganda. This project helps AIDS victims by providing schooling, medical attention and care made to measure. His home page is *http://www.moolenaar.net*, and his email address is `Bram@moolenaar.net`.

[S. Korea Gov to Use HancomLinux] Jan 9, 2002, South Korean government stepped into GNU/Linux desktop mainstream —120K copies desktop packages are to be supplied in this year by the contract between HancomLinux, a company developing and distributing GNU/Linux distribution and applications.
