Some interesting tools

use them or don't use them

LIA
Universidade de Vigo
Escola Superior de Enxeñaría Informática
E–32004 Ourense

http://lia.ei.uvigo.es
mailto:formella@uvigo.es

Contact: Arno Formella

Reference:	LIA-DOC-tools
Version:	1.3
Date:	25/11/2015
Pages:	37

Contents

1 Introduction

This document describes in a ready-to-use manner very convenient tools for the daily work. All scripts and configurations files can be downloaded from our web-site `http://lia.ei.uvigo.es`. Comments are welcome.

1.1 Purpose and scope

Although the descriptions in this document are tailored for a GNU/Linux environment, almost all tools are available for Windows and Mac systems as well.

Whenever you install a new working environment you should consider making these tools available to the user. The installation process is not included here, each tool describes the installation process in its own documentation.

The text is neither a tutorial nor a handbook; there are other sources for this type of documents. Rather the document describes a complete minimum setup of the tools which in most typical environments is sufficient to start with. It is clearly highlighted which parts must be adapted to fit personal needs.

Especially when you are working with or within the **LIA** research group, you should know and use these tools.

1.2 Revision history

Version 1.3:

- Markdown usage added.
- Usage of mount points for sharing file trees in various **`git`**-managed repositories added.

Version 1.2:

- Sections for **`dar`** (backup) recovering added.
- Section for **`make`** (task automation) added.

Version 1.1:

- Section for formatting C–source files added.
- Error in `cd`–command in setting up an empty repository in the **`git`** section fixed.

Version 1.0:
This is the initial document written in summer of 2010.

1.3 References

2 **dar**: Backup

Hardware fails, that is a fact. Therefore, you must backup at least all files that cannot be restored by other means, i.e., escentially you must backup your own work. This section briefly describes how to use the disk archiving tool **dar** to make first a full backup and then differential backups of your home directory. As principal backup media, we assume an external USB harddrive.

First of all, doing a good backup is not just copying files, rather you want to be able to restore a true snapshot of a file system including correct ownership for the files, correct time stamps, soft- and hardlinks, and much more. Hence, you need a tool, for instance **dar**.

dar, available at `http://dar.linux.free.fr` or `http://dar.sourceforge.net` is a GPL (v2) licensed software package to make backups of entire directory trees. **dar** is a shell command tool. There are GUIs available. The documentation is extensive and well done. **dar** can be tailored to make effective backups of entire systems. The main author is Denis Corbin, this document deals with version 2.3.10.

In order to make your backups, in the on-going you will see the most basic steps; feel free—with the help of the documentation of **dar**—to extend the suggestions.

● Note that we either store a full backup or a differential backup towards such a full backup, i.e., we do not iterate differential backups (a choice you can take if you like, **dar** is prepared for doing that).

2.1 Setup of the configuration file

dar can use a configuration file named `.darrc` in the users home directory (note, there are more options available to do the job). The listing below shows such a configuration file where we do the following:

- We divide the backup into slices of at most 600MB (`-s` option), so we can write them on CD whenever we need.
- We compress all files for which compression pays–off (`-z` and `-Z` options), so we save some disk space.
- We exclude temporary files and some files for security reasons (`-X` and `-P` options), so we save disk space and don't exhibit secrets.
- We make use of case-insensitive filenames and regular expressions. (`-an` and `-ar` options), so we are a bit more flexible.
- We maintain excluded directories as empty directories (`-D` option), so at least we know there has been something.
- We restore without any option (default run), so at least this will be easy after a crash.

Lines starting with a `#` are comment lines. The entries are sorted within groups in alphabetic order.

```
#
# (c) copyright 2010 Arno Formella
# formella@ei.uvigo.es, lia.ei.uvigo.es
#
```

```
# we run dar normally
#   to generate a full backup
#      dar -c ${1?}_`date -I`_full -R ${HOME?}
#   to generate a  differential backup
#      dar -c ${1?}_`date -I`_diff -R ${HOME?} -A ${2?}
#   to generate a catalog
#      dar -C ${1?}_`date -I`_ctlg -A ${2?}

# options for all command variants
all:
# slice always so we can grab on CD-ROM if we like
  -s 600M

# -c option, creation of an archive
create:
# compress with gzip
  -z
# but exclude from compression the following file types
# (they are usually compressed well enough)
# take the extension case-insensitive
  -an
  -Z "*.asf"
  -Z "*.avi"
  -Z "*.bz"
  -Z "*.bzip"
  -Z "*.bzip2"
  -Z "*.bz2"
  -Z "*.deb"
  -Z "*.divx"
  -Z "*.fdf"
  -Z "*.gif"
  -Z "*.gz"
  -Z "*.gzip"
  -Z "*.jar"
  -Z "*.jpg"
  -Z "*.mov"
  -Z "*.mp3"
  -Z "*.mp4"
  -Z "*.mpeg"
  -Z "*.mpg"
  -Z "*.ogg"
  -Z "*.pdf"
  -Z "*.rar"
  -Z "*.rpm"
  -Z "*.swf"
  -Z "*.sxw"
  -Z "*.tbz"
  -Z "*.tgz"
  -Z "*.tif"
  -Z "*.tiff"
  -Z "*.wma"
  -Z "*.wmv"
  -Z "*.zip"
  -Z "*.7z"
  -Z "*.Z"
  -acase
# create empty directories for excluded directories
  -D
# exclude the following configuration directories of certain
# applications, caches etc.
# because we don't care, after a crash we would
# install the applications anyway from scratch
  -P ".adobe/"
  -P ".agdserver/"
  -P ".fontconfig/"
  -P ".googleearth/"
  -P ".kde/"
  -P ".local/share/Trash/"
  -P ".loki/"
  -P ".macromedia/"
  -P ".mcop/"
  -P ".openoffice.org2/"
  -P ".qt/"
  -P ".strigi/"
  -P ".thumbnails/"
# exclude some things for security reasons
  -P ".ssh/"
  -X ".bash_history"
  -X ".forward"
  -X ".ICEauthority"
  -X ".lesshst"
  -X ".netrc"
  -X ".profile"
  -X ".recently-used"
  -X ".recently-used.xbel"
  -X ".sudo_as_admin_successful"
  -X ".viminfo"
  -X ".Xauthority"
  -X ".xsession_errors"
# exclude temporary files and automatically generated
# documentation files as well as all temporary files of
# compilers and the tex system
  -X "diff.txt"
  -X "err.txt"
  -X "ls.txt"
  -X "*~"
  -X "*.bak"
  -X "*.swp"
  -X ".gnuplot_history"
# here we switch to regular expression parsing for the options
  -ar
  -P ".*/\._d/"
  -P ".*/Cache/"
  -P ".*/\.libs/"
  -P ".*/doc/html/"
  -P ".*/doc/latex/"
# we switch back to normal file globbing
  -ag
  -X "*.a"
  -X "*.aux"
  -X "*.blg"
  -X "*.d"
  -X "*.dar"
  -X "*.dvi"
  -X "*.fls"
  -X "*.glg"
  -X "*.glo"
  -X "*.gls"
  -X "*.haux"
  -X "*.htoc"
  -X "*.idx"
  -X "*.ilg"
  -X "*.ind"
  -X "*.ist"
  -X "*.lo"
  -X "*.lod"
  -X "*.lof"
  -X "*.log"
  -X "*.lot"
  -X "*.maf"
  -X "*.mlf*"
  -X "*.mlt*"
  -X "*.mtc*"
  -X "*.nav"
  -X "*.o"
  -X "*.out"
  -X "*.plf*"
  -X "*.ptc*"
  -X "*.slf*"
  -X "*.slt*"
  -X "*.snm"
  -X "*.so"
  -X "*.stc*"
  -X "*.toc"
  -X "*.vrb"
# SRTM terrain files a huge and usually backuped somewhere else
  -X "*.hgt"
  -X "*.hgt.zip"
# exclude dar archives and iso-CD-burn files, too
  -X "*.*.dar"
  -X "*.iso"

# -x option, extraction of the files from an archive
# no options, it should work as simple as possible
extract:
```

- Feel free to modify the file patterns so they match your specific needs.

2.2 Invocation of `dar`

2.2.1 Full backup

Assume you want to backup your home directory on machine `machine` to an external USB drive mounted at `/media/disk` in the already existing directory `backups/machine`. To make a full backup and a lookup catalog, we run the command

```
source dar_bck_full.sh /media/disk/backups/machine/machine
```

with the following script `dar_bck_full.sh`

```
#
# (c) copyright 2010 Arno Formella
# formella@ei.uvigo.es, lia.ei.uvigo.es
#

# generate full backup
dar -c ${1?}_`date -I`_full -R ${HOME?}
# generate catalog
dar -C ${1?}_`date -I`_ctlg -A ${1?}_`date -I`_full
```

So we generate all slices of the full backup archive in the directory
`/media/disk/backups/machine`.
The slices are named, e.g.,
`machine_2008-08-28_full.1.dar`, `machine_2008-08-28_full.2.dar`, etc.,
and the catalog for faster future differential backups is named, e.g.,
`machine_2008-08-28_ctlg.1.dar`.

2.2.2 Differential backup

To make a differential backup, we run the command

```
source dar_bck_diff.sh \
  /media/disk/backups/machine/machine \
  /media/disk/backups/machine/machine_YYYY-MM-DD_ctlg
```

where you have to substitute the sequence `YYYY-MM-DD` according to the corresponding date of the base full backup, with the following script `dar_bck_diff.sh`

```
#
# (c) copyright 2010 Arno Formella
# formella@ei.uvigo.es, lia.ei.uvigo.es
#

# generate differential backup
dar -c ${1?}_`date -I`_diff -R ${HOME?} -A ${2?}
```

2.2.3 Restoring the backup

To restore a backup back onto your system just use twice the script `dar_bck_rest.sh`

```
#
# (c) copyright 2010 Arno Formella
# formella@ei.uvigo.es, lia.ei.uvigo.es
#

# restore backup to home directory
dar -x -w -r ${1?} -R ${HOME?}
```

where the first time you specify as argument the base name of the full backup and the second time the base name of the differential backup. The additional options `-w` means to overwrite files but `-r` confines the writing to overwriting older files by newer ones, but not the other way round.

2.2.4 Restoring some files only

To restore or recover a specific file from your backup (for instance, in case it was deleted/lost for some reason) run the script `dar_bck_reco.sh`

```
#
# (c) copyright 2010 Arno Formella
# formella@ei.uvigo.es, lia.ei.uvigo.es
#

# restore backup to home directory
dar -R ${HOME} -x ${1?} -v -g ${2?}
```

with the first argument specifing the differential backup and the second argument specifying the file to be recovered with its relative path name starting from your home directory. If you see a line like

```
restoring file: /home/denis/my_precious_file
```

you are done, otherwise run the script again but now with the full backup as first argument.

For more sophisticated recovering actions, please see the manual of **dar**and consider using `dar_manager`.

3 **git**: Version control

Version control (or revision control) and backup are in certain aspects quite similar, however, their main objectives are different: the main purpose of a backup is to be able to recover smoothly after a more or less severe crash; version control means that you can go back to certain points in time and restore the ancient snapshot of a development, possibly maintaining different branches within the same project.

There are many version control software tools available. **git** is one of them with one outstanding characteristic: the possibility of serverless distributed colaboration.

git, available at `http://www.git-scm.com` is a GPL (v2) licensed software package for fast, scalable, distributed revision control. **git** is a shell command tool. There are graphical user interfaces (GUIs) available. The documentation is extensive and well done. There are migration tools to/from other version control systems available. The main author is Linus Torvalds, this document was written taking into account version 1.7.2.1, maintainer Junio C. Hamano. Note that **git** is evolving.

In order to control your devolopment, in the on-going you will see the most basic setups and simple scripts; feel free—with the help of the documentation of **git**—to extend the suggestions.

3.1 Initialization of **git**

The initialization of **git** is—at least for a minimum configuration—quite simple. There are some globally available settings to be done (you can even skip this part, then **git** will guess the settings from your system and user installation). The most basic are telling **git** your user name and your email address, so the repository will have this information ready for other users possibly working with it. Here we use global settings on the machine (You might want to use different data in different repositories, which is possible as well.)

```
git config --global user.name <your name>
git config --global user.email <your email>
```

The **git**–command stores such configuration data in a user local configuration file named `.gitconfig` being placed in your home directory. See the **git** documentation for more details about the possible content of this file.

To proceed change the directory to your working directory and initialize a new empty **git** repository:

```
cd <workdir>
git init
```

This directory can be an empty directory when you just start a new project, or it can be an already populated one that you want to put under **git** version control. The command generates a subdirectory named `.git` and places some intial configuration data in there. Later the directory will contain the entire repository for the project. Mostly, you don't have to deal directly with the content of this directory, however, make sure that it is part of your backup.

Edit in the working directory a `.gitignore`–file which will contain all file patterns to be ignored by **git**. An simple example of a `.gitignore`–file for a directory used for C++ programming is:

```
#
```

```
# (c) copyright 2010 Arno Formella
# formella@ei.uvigo.es, lia.ei.uvigo.es
#

# Note: take care of trailing white space characters at end
# of pattern git does not ignore them and you may get
# unexpected effects.

# the compiler generated files
*.a
*.d
*.o
*.lo
*.so
*.Tpo

# temporary files or links for compilation
.depend
.make.state
CONFIG

# the entire documentation files, but the logos
*/src/doc/latex/
*/src/doc/html/*
*/src/doc/html/*/
!*/src/doc/html/doxy_*.png

# backup files of different tools
*~
*.bak
*.swp

# results of diff and error output, I usually use
diff.txt
err.txt

# archive files
*.deb
*.tar
*.tar.bz*
*.tar.gz
*.tbz
*.tgz

# autoconf configuration files
config.mak
autom4te.cache
config.cache
config.log
config.status
config.mak.autogen
config.mak.append
configure
```

As you see, lines starting with the #–symbol serve as comment lines. You can use wildcards as usual. An example of a `.gitignore`–file for a LaTeX text working directory is:

```
#
# (c) copyright 2010 Arno Formella
# formella@ei.uvigo.es, lia.ei.uvigo.es
#

# Note: take care of trailing white space characters at end
# of pattern git does not ignore them and you may get
# unexpected effects.

# Ignore all files latex tools generate as output.
# Usually these files can be re-generated
# with the exception of implicite dates in the output.
# So take care that you use absolute dates if a future
# re-run of latex should produce exactly the same output
*.bbl
*.dvi
*.gls
*.html
*.ind
*.pdf
*.ps

# Do not ignore scanned, signed, or marked final pdf files.
!*_final.pdf
!*_scanned.pdf
!*_signed.pdf

# Ignore all temporary files generated by a latex system.
*.aux
*.blg
*.fls
*.glg
*.glo
*.haux
*.htoc
*.idx
*.ilg
*.ist
*.lod
*.lof
*.log
*.lot
*.maf
*.mlf*
*.mlt*
*.mtc*
*.nav
*.out
*.plf*
*.plt*
*.ptc*
*.slf*
*.slt*
```

```
*.snm
*.stc*
*.toc
*.vrb

# Ignore tree of make tool
._d/
```

● Note that you must not use trailing white spaces in the patterns, because **git** will interpret these characters as part of the pattern which may not be what you want.

● Note that the excluded files and directories should be related to your usual way of working and should be adapted to your needs and the tools you employ.

3.2 Working with **git**

This section briefly describes how to put files under **git** control and how to commit modifications of the files such that they are recorded in the repository.

To put all files that do not match any pattern in your `.gitignore` file under control of **git** execute:

```
git add .
git commit -a
```

The `add`–command adds all modified/created files to the index (marking files to be handled) and the `commit`–command stores the necessary information into the repository. Instead of specifying all changed and non-excluded files, you can work with individual files as well:

```
git add <filename>
git commit <filename>
```

3.3 Using **git** to move your files from system to system

We use **git** to maintain a repository on a USB-memory pen drive which serves to transfer files from one computer to another.

3.3.1 Preparing a USB memory pen drive

You might want to install an `ext3` file system on the pen drive and give it a name. In most cases, such devices are delivered with a `FAT` file system. You can skip this formating of the pen drive, however, a GNU/Linux file system has certain advantages over a `FAT` system.

● Be aware that you will loose all information stored on the device, so take precautions beforehand.

Plug in your pen drive in a free slot. Once the device is mounted run

```
df
```

In the listing that appears the first column indicates the device and the last column the directory name where the device is mounted. Find your pen drive in the listing (a good candidate is `/media/disk` which is used below in the examples).

Unmount the device:

```
sudo umount /media/disk
```

Format the device, give it a name, and synchronize the system:

```
sudo mkfs.ext3 /dev/sdb1
sudo e2label /dev/sdb1 liadisk
sync
```

● **Warning:** be careful to use the correct device, otherwise you may destroy relevant data on other devices.

Finally you can re-mount the device, or just remove and re-plug-in.

3.3.2 Setting up an empty repository

● First setup a bare repository on the pen drive. As example we use here the repository name `lia.git`. Note that **git**-repositories by convention use the `.git`-extension. We assume that an `ext3` file system with root previleges has been installed (see previous section), hence, we create the directory of the repository as root and change its mode so all users can act on it.

```
sudo mkdir /media/liadisk/lia.git
sudo chmod -R 0777 /media/liadisk/lia.git
cd /media/liadisk/lia.git
git --bare init
```

3.3.3 Cloning the empty repository onto your computers

Clone the empty **git** repository that we created on the pen drive onto one of your computers (and ignore the warning emitted by **git**):

```
git clone /media/liadisk/lia.git
```

This will create at the point in the file system where you run the command a directory named `lia` containing just the `.git` subtree, but still no more files.

● Either start a new project there or populate the directory with files your want to put under **git** control. Don't forget to include an appropriate `.gitignore`–file. Work as usual with **git** adding and committing files. Once everything is finished, push the changes to the pen drive, for instance with:

```
<work-with-files>
git add .
git commit -a
git push origin master
```

This will update the repository on the pen drive with your local repository content, i.e., you will have an exact copy.

```
git clone /media/liadisk/lia.git
<work-with-files>
git add .
git commit -a
git push origin master
```

This again will update the repository in the pen drive with your local repository on the second machine.

From now on, you work always in the following way:

- Plug-in your pen drive.
- Pull the repository from the pen on your computer. If nothing has been modified, you get the appropriate message. If you forgot to plug-in your pen drive, you will get the appropriate error message. If you must merge, you are prompted to do so.
- Work with your local repository as usual with **`git`**.
- Push the changes to the pen drive.

The whole process with **`git`**–commands (as example, there is more...):

```
git pull
<work-with-files>
git add .
git commit -a
git push origin master
```

Note that we created three identical repositories. We can either work, as described, only on one computer at a time using the pen drive to synchronize whenever we switch machine, or we can develop on the different machines, possibly even with different users, different branches of the project which can be merged—at the points whereever we wish (see **`git`** documentation for more details of these more complex operations).

It may be convenient to hold at each location a bare repository as well, so you can push from anywhere to this bare repository and synchronize the local working repository whenever you work on that location again.

3.4 Tips and tricks for **`git`**

3.4.1 Mount points

If you want to store a file tree in various projects each one with its own repository managed with **`git`** but without duplicating the files, hence, you work always with the very same files in both projects but you share them in different projects (possibly with different people), you can consider mounting directories.

For example, if you have two projects sharing the source code of some library, i.e., the `.git` directories are located under the project folder:

```
project_one/.git/...
           /some_lib/src/...
project_two/.git/...
           /some_lib/src/...
```

you can remove all files below the directory content to be shared in the second project and mount the corresponding directory of the first project, e.g.,

```
cd project_two/some_lib/src
rm *
cd ..
sudo mount -B ../../project_one/some_lib/src src
```

If you want to avoid the `sudo`-command look into the corresponding man pages. Here, only the `src`-directory is shared, the `some_lib`-directory might contain project specific files.

● Note that you take care to exclude the mount points in your backup system if necessary. Many backup systems do not handle the mount point correctly, rather they will store/restore the file tree twice.

4 **`make`**: Task automation

4.1 Makefile for LaTeX documents

4.2 Makefile for C++ compilation

5 **markdown**: Simple formatted text

markdown (http://daringfireball.net/projects/markdown) is a simple text formatting syntax which layouts the text with very few special symbols in such a way that it is easily readable in source and can be converted into other markup languages, such as HTML.

markdown is supported by **doxygen** (http://www.stack.nl/~dimitri/doxygen/manual/markdown.html), so it is a convenient and easy to use tool in order to format the documentation of the source code for several programming languages.

markdown can be used in LaTeX–documents as well, as explained in the on-going.

5.1 Embedding **markdown** in LaTeX

One possibility to include markdown formatted text into a LaTeX-document is to use **pandoc** (http://johnmacfarlane.net/pandoc) as converting tool called from a special environment as shown here (this suggestion is taken from an answer given by G. Poore at http://tex.stackexchange.com/questions/101717/converting-markdown-to-latex-in-latex:

```
\documentclass{article}

\usepackage{ctable}
\usepackage{fancyvrb}

\newenvironment{markdown}%
{\VerbatimEnvironment\begin{VerbatimOut}{tmp.markdown}}%
{\end{VerbatimOut}%
 \immediate\write18{pandoc tmp.markdown -t latex -o tmp.tex}%
 \input{tmp.tex}%
}

\begin{document}

\begin{markdown}
## Example

Some text that goes on for a while.

A list:

* Item
* Another item

A table:

Program  Name
----     ----
psm      point set match
hms      HumSAT mission software

\end{markdown}
```

```
\end{document}
```

Note the import of the two packages `ctable` and `fancyvrb`. Make sure to use **pandoc** version 1.6.0 or higher, so deeper section levels are handled correctly as well.

Assuming that above `.tex`–file is named `example.tex`, you generate the `.pdf`–output with the command:

```
pdflatex --shell-escape example.tex
```

The basic syntax of **markdown** is given at `http://daringfireball.net/projects/markdown/syntax`. Section 5.3 gives an introduction taken from there. **markdown** allows also to embed directly HTML tags, which might be necessary in a certain context—for instance while documenting code—to introduce advanced formatting. Note that these HTML–constructs possibly will not be adequately translated to LaTeX; this depends on the capabilities of the **pandoc** version you use.

markdown provides support for headlines or titles, blockquotes, lists (bulleted or numbered), code blocks (written in typewriter font), horizontal rules, hyperlinks, text highlighting (italics and bold), and some rudimentary image including.

The following section shows how the above **markdown** environment is converted to LaTeX.

5.2 Example

Some text that goes on for a while.

A list:

- Item
- Another item

A table:

Program	Name
psm	point set match
hms	HumSAT mission software

5.3 **markdown**: Syntax

5.3.1 Overview

Philosophy

Markdown is intended to be as easy-to-read and easy-to-write as is feasible.

Readability, however, is emphasized above all else. A Markdown-formatted document should be publishable as-is, as plain text, without looking like it's been marked up with tags or formatting instructions. While Markdown's syntax has been influenced by several existing text-to-HTML filters — including Setext, atx, Textile, reStructuredText, Grutatext, and EtText — the single biggest source of inspiration for Markdown's syntax is the format of plain text email.

To this end, Markdown's syntax is comprised entirely of punctuation characters, which punctuation characters have been carefully chosen so as to look like what they mean. E.g., asterisks around a word actually look like *emphasis*. Markdown lists look like, well, lists. Even blockquotes look like quoted passages of text, assuming you've ever used email.

Inline HTML

Markdown's syntax is intended for one purpose: to be used as a format for *writing* for the web.

Markdown is not a replacement for HTML, or even close to it. Its syntax is very small, corresponding only to a very small subset of HTML tags. The idea is *not* to create a syntax that makes it easier to insert HTML tags. In my opinion, HTML tags are already easy to insert. The idea for Markdown is to make it easy to read, write, and edit prose. HTML is a *publishing* format; Markdown is a *writing* format. Thus, Markdown's formatting syntax only addresses issues that can be conveyed in plain text.

For any markup that is not covered by Markdown's syntax, you simply use HTML itself. There's no need to preface it or delimit it to indicate that you're switching from Markdown to HTML; you just use the tags.

The only restrictions are that block-level HTML elements — e.g. `<div>`, `<table>`, `<pre>`, `<p>`, etc. — must be separated from surrounding content by blank lines, and the start and end tags of the block should not be indented with tabs or spaces. Markdown is smart enough not to add extra (unwanted) `<p>` tags around HTML block-level tags.

For example, to add an HTML table to a Markdown article:

```
This is a regular paragraph.

<table>
    <tr>
        <td>Foo</td>
    </tr>
</table>

This is another regular paragraph.
```

Note that Markdown formatting syntax is not processed within block-level HTML tags. E.g., you can't use Markdown-style `*emphasis*` inside an HTML block.

Span-level HTML tags — e.g. `<span>`, `<cite>`, or `<del>` — can be used anywhere in a Markdown paragraph, list item, or header. If you want, you can even use HTML tags instead of Markdown formatting; e.g. if you'd prefer to use HTML `<a>` or `<img>` tags instead of Markdown's link or image syntax, go right ahead.

Unlike block-level HTML tags, Markdown syntax *is* processed within span-level tags.

Automatic Escaping for Special Characters

In HTML, there are two characters that demand special treatment: < and `&`. Left angle brackets are used to start tags; ampersands are used to denote HTML entities. If you want to use them as literal characters, you must escape them as entities, e.g. `&lt;`, and `&amp;`.

Ampersands in particular are bedeviling for web writers. If you want to write about 'AT&T', you need to write '`AT&amp;T`'. You even need to escape ampersands within URLs. Thus, if you want to link to:

```
http://images.google.com/images?num=30&q=larry+bird
```

you need to encode the URL as:

```
http://images.google.com/images?num=30&amp;q=larry+bird
```

in your anchor tag `href` attribute. Needless to say, this is easy to forget, and is probably the single most common source of HTML validation errors in otherwise well-marked-up web sites.

Markdown allows you to use these characters naturally, taking care of all the necessary escaping for you. If you use an ampersand as part of an HTML entity, it remains unchanged; otherwise it will be translated into `&amp;`.

So, if you want to include a copyright symbol in your article, you can write:

```
&copy;
```

and Markdown will leave it alone. But if you write:

```
AT&T
```

Markdown will translate it to:

```
AT&amp;T
```

Similarly, because Markdown supports inline HTML, if you use angle brackets as delimiters for HTML tags, Markdown will treat them as such. But if you write:

```
4 < 5
```

Markdown will translate it to:

```
4 &lt; 5
```

However, inside Markdown code spans and blocks, angle brackets and ampersands are *always* encoded automatically. This makes it easy to use Markdown to write about HTML code. (As opposed to raw HTML, which is a terrible format for writing about HTML syntax, because every single < and `&` in your example code needs to be escaped.)

5.3.2 Block Elements

Paragraphs and Line Breaks

A paragraph is simply one or more consecutive lines of text, separated by one or more blank lines. (A blank line is any line that looks like a blank line — a line containing nothing but spaces or tabs is considered blank.) Normal paragraphs should not be indented with spaces or tabs.

The implication of the “one or more consecutive lines of text” rule is that Markdown supports “hard-wrapped” text paragraphs. This differs significantly from most other text-to-HTML formatters (including Movable Type’s “Convert Line Breaks” option) which translate every line break character in a paragraph into a `<br />` tag.

When you *do* want to insert a `<br />` break tag using Markdown, you end a line with two or more spaces, then type return.

Yes, this takes a tad more effort to create a `<br />`, but a simplistic "every line break is a `<br />`" rule wouldn't work for Markdown. Markdown's email-style blockquoting and multi-paragraph list items work best — and look better — when you format them with hard breaks.

Headers

Markdown supports two styles of headers, Setext and atx.

Setext-style headers are "underlined" using equal signs (for first-level headers) and dashes (for second-level headers). For example:

```
This is an H1
=============

This is an H2
-------------
```

Any number of underlining =’s or −’s will work.

Atx-style headers use 1–6 hash characters at the start of the line, corresponding to header levels 1–6. For example:

```
# This is an H1

## This is an H2

###### This is an H6
```

Optionally, you may "close" atx-style headers. This is purely cosmetic — you can use this if you think it looks better. The closing hashes don't even need to match the number of hashes used to open the header. (The number of opening hashes determines the header level.) :

```
# This is an H1 #

## This is an H2 ##

### This is an H3 ######
```

Blockquotes

Markdown uses email-style > characters for blockquoting. If you're familiar with quoting passages of text in an email message, then you know how to create a blockquote in Markdown. It looks best if you hard wrap the text and put a > before every line:

```
> This is a blockquote with two paragraphs. Lorem ipsum dolor sit amet,
> consectetuer adipiscing elit. Aliquam hendrerit mi posuere lectus.
> Vestibulum enim wisi, viverra nec, fringilla in, laoreet vitae, risus.
>
> Donec sit amet nisl. Aliquam semper ipsum sit amet velit. Suspendisse
> id sem consectetuer libero luctus adipiscing.
```

Markdown allows you to be lazy and only put the > before the first line of a hard-wrapped paragraph:

```
> This is a blockquote with two paragraphs. Lorem ipsum dolor sit amet,
consectetuer adipiscing elit. Aliquam hendrerit mi posuere lectus.
Vestibulum enim wisi, viverra nec, fringilla in, laoreet vitae, risus.

> Donec sit amet nisl. Aliquam semper ipsum sit amet velit. Suspendisse
id sem consectetuer libero luctus adipiscing.
```

Blockquotes can be nested (i.e. a blockquote-in-a-blockquote) by adding additional levels of >:

```
> This is the first level of quoting.
>
> > This is nested blockquote.
>
> Back to the first level.
```

Blockquotes can contain other Markdown elements, including headers, lists, and code blocks:

```
    > ## This is a header.
    >
    > 1.   This is the first list item.
    > 2.   This is the second list item.
    >
    > Here's some example code:
    >
    >     return shell_exec("echo $input | $markdown_script");
```

Any decent text editor should make email-style quoting easy. For example, with BBEdit, you can make a selection and choose Increase Quote Level from the Text menu.

Lists

Markdown supports ordered (numbered) and unordered (bulleted) lists.

Unordered lists use asterisks, pluses, and hyphens — interchangably — as list markers:

```
*   Red
*   Green
*   Blue
```

is equivalent to:

```
+   Red
+   Green
+   Blue
```

and:

```
-   Red
-   Green
-   Blue
```

Ordered lists use numbers followed by periods:

```
1.  Bird
2.  McHale
3.  Parish
```

It's important to note that the actual numbers you use to mark the list have no effect on the HTML output Markdown produces. The HTML Markdown produces from the above list is:

```
<ol>
<li>Bird</li>
<li>McHale</li>
<li>Parish</li>
</ol>
```

If you instead wrote the list in Markdown like this:

```
1.  Bird
1.  McHale
1.  Parish
```

or even:

```
3. Bird
1. McHale
8. Parish
```

you'd get the exact same HTML output. The point is, if you want to, you can use ordinal numbers in your ordered Markdown lists, so that the numbers in your source match the numbers in your published HTML. But if you want to be lazy, you don't have to.

If you do use lazy list numbering, however, you should still start the list with the number 1. At some point in the future, Markdown may support starting ordered lists at an arbitrary number.

List markers typically start at the left margin, but may be indented by up to three spaces. List markers must be followed by one or more spaces or a tab.

To make lists look nice, you can wrap items with hanging indents:

```
*   Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
    Aliquam hendrerit mi posuere lectus. Vestibulum enim wisi,
    viverra nec, fringilla in, laoreet vitae, risus.
*   Donec sit amet nisl. Aliquam semper ipsum sit amet velit.
    Suspendisse id sem consectetuer libero luctus adipiscing.
```

But if you want to be lazy, you don't have to:

```
*   Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
Aliquam hendrerit mi posuere lectus. Vestibulum enim wisi,
viverra nec, fringilla in, laoreet vitae, risus.
*   Donec sit amet nisl. Aliquam semper ipsum sit amet velit.
Suspendisse id sem consectetuer libero luctus adipiscing.
```

If list items are separated by blank lines, Markdown will wrap the items in `<p>` tags in the HTML output. For example, this input:

```
*   Bird
*   Magic
```

will turn into:

```
<ul>
<li>Bird</li>
<li>Magic</li>
</ul>
```

But this:

```
*   Bird

*   Magic
```

will turn into:

```
<ul>
<li><p>Bird</p></li>
<li><p>Magic</p></li>
</ul>
```

List items may consist of multiple paragraphs. Each subsequent paragraph in a list item must be indented by either 4 spaces or one tab:

```
1.  This is a list item with two paragraphs. Lorem ipsum dolor
    sit amet, consectetuer adipiscing elit. Aliquam hendrerit
    mi posuere lectus.

    Vestibulum enim wisi, viverra nec, fringilla in, laoreet
    vitae, risus. Donec sit amet nisl. Aliquam semper ipsum
    sit amet velit.

2.  Suspendisse id sem consectetuer libero luctus adipiscing.
```

It looks nice if you indent every line of the subsequent paragraphs, but here again, Markdown will allow you to be lazy:

```
*   This is a list item with two paragraphs.

    This is the second paragraph in the list item. You're
only required to indent the first line. Lorem ipsum dolor
sit amet, consectetuer adipiscing elit.

*   Another item in the same list.
```

To put a blockquote within a list item, the blockquote's > delimiters need to be indented:

```
*   A list item with a blockquote:

    > This is a blockquote
    > inside a list item.
```

To put a code block within a list item, the code block needs to be indented *twice* — 8 spaces or two tabs:

```
*   A list item with a code block:

        <code goes here>
```

It's worth noting that it's possible to trigger an ordered list by accident, by writing something like this:

```
1986. What a great season.
```

In other words, a *number-period-space* sequence at the beginning of a line. To avoid this, you can backslash-escape the period:

```
1986\. What a great season.
```

Code Blocks

Pre-formatted code blocks are used for writing about programming or markup source code. Rather than forming normal paragraphs, the lines of a code block are interpreted literally. Markdown wraps a code block in both `<pre>` and `<code>` tags.

To produce a code block in Markdown, simply indent every line of the block by at least 4 spaces or 1 tab. For example, given this input:

```
This is a normal paragraph:

    This is a code block.
```

Markdown will generate:

```
<p>This is a normal paragraph:</p>

<pre><code>This is a code block.
</code></pre>
```

One level of indentation — 4 spaces or 1 tab — is removed from each line of the code block. For example, this:

```
Here is an example of AppleScript:

    tell application "Foo"
        beep
    end tell
```

will turn into:

```
<p>Here is an example of AppleScript:</p>

<pre><code>tell application "Foo"
    beep
end tell
</code></pre>
```

A code block continues until it reaches a line that is not indented (or the end of the article).

Within a code block, ampersands (`&`) and angle brackets (< and >) are automatically converted into HTML entities. This makes it very easy to include example HTML source code using Markdown — just paste it and indent it, and Markdown will handle the hassle of encoding the ampersands and angle brackets. For example, this:

```
    <div class="footer">
        &copy; 2004 Foo Corporation
    </div>
```

will turn into:

```
<pre><code>&lt;div class="footer"&gt;
    &amp;copy; 2004 Foo Corporation
&lt;/div&gt;
</code></pre>
```

Regular Markdown syntax is not processed within code blocks. E.g., asterisks are just literal asterisks within a code block. This means it's also easy to use Markdown to write about Markdown's own syntax.

Horizontal Rules

You can produce a horizontal rule tag (`<hr />`) by placing three or more hyphens, asterisks, or underscores on a line by themselves. If you wish, you may use spaces between the hyphens or asterisks. Each of the following lines will produce a horizontal rule:

```
* * *

***

*****

- - -

---------------------------------------
```

5.3.3 Span Elements

Links

Markdown supports two style of links: *inline* and *reference*.

In both styles, the link text is delimited by [square brackets].

To create an inline link, use a set of regular parentheses immediately after the link text's closing square bracket. Inside the parentheses, put the URL where you want the link to point, along with an *optional* title for the link, surrounded in quotes. For example:

```
This is [an example](http://example.com/ "Title") inline link.

[This link](http://example.net/) has no title attribute.
```

Will produce:

```
<p>This is <a href="http://example.com/" title="Title">
an example</a> inline link.</p>

<p><a href="http://example.net/">This link</a> has no
title attribute.</p>
```

If you're referring to a local resource on the same server, you can use relative paths:

```
See my [About](/about/) page for details.
```

Reference-style links use a second set of square brackets, inside which you place a label of your choosing to identify the link:

```
This is [an example][id] reference-style link.
```

You can optionally use a space to separate the sets of brackets:

```
This is [an example] [id] reference-style link.
```

Then, anywhere in the document, you define your link label like this, on a line by itself:

```
[id]: http://example.com/  "Optional Title Here"
```

That is:

- Square brackets containing the link identifier (optionally indented from the left margin using up to three spaces);
- followed by a colon;
- followed by one or more spaces (or tabs);
- followed by the URL for the link;
- optionally followed by a title attribute for the link, enclosed in double or single quotes, or enclosed in parentheses.

The following three link definitions are equivalent:

```
    [foo]: http://example.com/  "Optional Title Here"
    [foo]: http://example.com/  'Optional Title Here'
    [foo]: http://example.com/  (Optional Title Here)
```

Note: There is a known bug in Markdown.pl 1.0.1 which prevents single quotes from being used to delimit link titles.

The link URL may, optionally, be surrounded by angle brackets:

```
[id]: <http://example.com/>  "Optional Title Here"
```

You can put the title attribute on the next line and use extra spaces or tabs for padding, which tends to look better with longer URLs:

```
[id]: http://example.com/longish/path/to/resource/here
    "Optional Title Here"
```

Link definitions are only used for creating links during Markdown processing, and are stripped from your document in the HTML output.

Link definition names may consist of letters, numbers, spaces, and punctuation — but they are *not* case sensitive. E.g. these two links:

```
    [link text][a]
    [link text][A]
```

are equivalent.

The *implicit link name* shortcut allows you to omit the name of the link, in which case the link text itself is used as the name. Just use an empty set of square brackets — e.g., to link the word "Google" to the google.com web site, you could simply write:

```
    [Google][]
```

And then define the link:

```
    [Google]: http://google.com/
```

Because link names may contain spaces, this shortcut even works for multiple words in the link text:

```
    Visit [Daring Fireball][] for more information.
```

And then define the link:

```
    [Daring Fireball]: http://daringfireball.net/
```

Link definitions can be placed anywhere in your Markdown document. I tend to put them immediately after each paragraph in which they're used, but if you want, you can put them all at the end of your document, sort of like footnotes.

Here's an example of reference links in action:

```
I get 10 times more traffic from [Google] [1] than from
[Yahoo] [2] or [MSN] [3].

  [1]: http://google.com/        "Google"
  [2]: http://search.yahoo.com/  "Yahoo Search"
  [3]: http://search.msn.com/    "MSN Search"
```

Using the implicit link name shortcut, you could instead write:

```
I get 10 times more traffic from [Google][] than from
[Yahoo][] or [MSN][].

  [google]: http://google.com/        "Google"
  [yahoo]:  http://search.yahoo.com/  "Yahoo Search"
  [msn]:    http://search.msn.com/    "MSN Search"
```

Both of the above examples will produce the following HTML output:

```
<p>I get 10 times more traffic from <a href="http://google.com/"
title="Google">Google</a> than from
<a href="http://search.yahoo.com/" title="Yahoo Search">Yahoo</a>
or <a href="http://search.msn.com/" title="MSN Search">MSN</a>.</p>
```

For comparison, here is the same paragraph written using Markdown's inline link style:

```
I get 10 times more traffic from [Google](http://google.com/ "Google")
than from [Yahoo](http://search.yahoo.com/ "Yahoo Search") or
[MSN](http://search.msn.com/ "MSN Search").
```

The point of reference-style links is not that they're easier to write. The point is that with reference-style links, your document source is vastly more readable. Compare the above examples: using reference-style links, the paragraph itself is only 81 characters long; with inline-style links, it's 176 characters; and as raw HTML, it's 234 characters. In the raw HTML, there's more markup than there is text.

With Markdown's reference-style links, a source document much more closely resembles the final output, as rendered in a browser. By allowing you to move the markup-related metadata out of the paragraph, you can add links without interrupting the narrative flow of your prose.

Emphasis

Markdown treats asterisks (*) and underscores (_) as indicators of emphasis. Text wrapped with one * or _ will be wrapped with an HTML `<em>` tag; double *'s or _'s will be wrapped with an HTML `<strong>` tag. E.g., this input:

```
*single asterisks*
```

```
_single underscores_

**double asterisks**

__double underscores__
```

will produce:

```
<em>single asterisks</em>

<em>single underscores</em>

<strong>double asterisks</strong>

<strong>double underscores</strong>
```

You can use whichever style you prefer; the lone restriction is that the same character must be used to open and close an emphasis span.

Emphasis can be used in the middle of a word:

```
un*frigging*believable
```

But if you surround an * or _ with spaces, it'll be treated as a literal asterisk or underscore.

To produce a literal asterisk or underscore at a position where it would otherwise be used as an emphasis delimiter, you can backslash escape it:

```
\*this text is surrounded by literal asterisks\*
```

Code

To indicate a span of code, wrap it with backtick quotes (`). Unlike a pre-formatted code block, a code span indicates code within a normal paragraph. For example:

```
Use the `printf()` function.
```

will produce:

```
<p>Use the <code>printf()</code> function.</p>
```

To include a literal backtick character within a code span, you can use multiple backticks as the opening and closing delimiters:

````
``There is a literal backtick (`) here.``
````

which will produce this:

```
<p><code>There is a literal backtick (`) here.</code></p>
```

The backtick delimiters surrounding a code span may include spaces — one after the opening, one before the closing. This allows you to place literal backtick characters at the beginning or end of a code span:

```
    A single backtick in a code span: `` ` ``

    A backtick-delimited string in a code span: `` `foo` ``
```

will produce:

```
    <p>A single backtick in a code span: <code>`</code></p>

    <p>A backtick-delimited string in a code span: <code>`foo`</code></p>
```

With a code span, ampersands and angle brackets are encoded as HTML entities automatically, which makes it easy to include example HTML tags. Markdown will turn this:

```
Please don't use any `<blink>` tags.
```

into:

```
<p>Please don't use any <code>&lt;blink&gt;</code> tags.</p>
```

You can write this:

```
`&#8212;` is the decimal-encoded equivalent of `&mdash;`.
```

to produce:

```
<p><code>&amp;#8212;</code> is the decimal-encoded
equivalent of <code>&amp;mdash;</code>.</p>
```

Images

Admittedly, it's fairly difficult to devise a "natural" syntax for placing images into a plain text document format.

Markdown uses an image syntax that is intended to resemble the syntax for links, allowing for two styles: *inline* and *reference*.

Inline image syntax looks like this:

```
![Alt text](/path/to/img.jpg)

![Alt text](/path/to/img.jpg "Optional title")
```

That is:

- An exclamation mark: `!`;

- followed by a set of square brackets, containing the `alt` attribute text for the image;
- followed by a set of parentheses, containing the URL or path to the image, and an optional `title` attribute enclosed in double or single quotes.

Reference-style image syntax looks like this:

```
![Alt text][id]
```

Where "id" is the name of a defined image reference. Image references are defined using syntax identical to link references:

```
[id]: url/to/image  "Optional title attribute"
```

As of this writing, Markdown has no syntax for specifying the dimensions of an image; if this is important to you, you can simply use regular HTML `<img>` tags.

5.3.4 Miscellaneous

Automatic Links

Markdown supports a shortcut style for creating "automatic" links for URLs and email addresses: simply surround the URL or email address with angle brackets. What this means is that if you want to show the actual text of a URL or email address, and also have it be a clickable link, you can do this:

```
<http://example.com/>
```

Markdown will turn this into:

```
<a href="http://example.com/">http://example.com/</a>
```

Automatic links for email addresses work similarly, except that Markdown will also perform a bit of randomized decimal and hex entity-encoding to help obscure your address from address-harvesting spambots. For example, Markdown will turn this:

```
<address@example.com>
```

into something like this:

```
<a href="&#x6D;&#x61;i&#x6C;&#x74;&#x6F;:&#x61;&#x64;&#x64;&#x72;&#x65;
&#115;&#115;&#64;&#101;&#120;&#x61;&#109;&#x70;&#x6C;e&#x2E;&#99;&#111;
&#109;">&#x61;&#x64;&#x64;&#x72;&#x65;&#115;&#115;&#64;&#101;&#120;&#x61;
&#109;&#x70;&#x6C;e&#x2E;&#99;&#111;&#109;</a>
```

which will render in a browser as a clickable link to “address@example.com”.

(This sort of entity-encoding trick will indeed fool many, if not most, address-harvesting bots, but it definitely won’t fool all of them. It’s better than nothing, but an address published in this way will probably eventually start receiving spam.)

Backslash Escapes

Markdown allows you to use backslash escapes to generate literal characters which would otherwise have special meaning in Markdown’s formatting syntax. For example, if you wanted to surround a word with literal asterisks (instead of an HTML `<em>` tag), you can use backslashes before the asterisks, like this:

```
\*literal asterisks\*
```

Markdown provides backslash escapes for the following characters:

```
\   backslash
`   backtick
*   asterisk
_   underscore
{}  curly braces
[]  square brackets
()  parentheses
#   hash mark
    +        plus sign
    -        minus sign (hyphen)
.   dot
!   exclamation mark
```

6 **indent**: formatting of C–source files

The classical **indent**command line program can be used to format C–source files (`.c` and .h files) more or less similar to the programming style of **LIA** for C++. You use the following indent profile (i.e., `.indent.pro` in your home directory).

```
//
// (c) copyright 2010 Arno Formella
// formella@ei.uvigo.es, lia.ei.uvigo.es
//

//      --blank-lines-after-commas                //      -bc
        --blank-lines-after-declarations          //      -bad
        --blank-lines-after-procedures            //      -bap
        --blank-lines-before-block-comments       //      -bbb
//      --braces-after-if-line                    //      -bl
        --braces-after-func-def-line              //      -blf
//      --brace-indent                            //      -bli
//      --braces-after-struct-decl-line           //      -bls
        --braces-on-if-line                       //      -br
        --braces-on-func-def-line                 //      -brf
        --braces-on-struct-decl-line              //      -brs
        --break-after-boolean-operator            //      -nbbo
//      --break-before-boolean-operator           //      -bbo
        --break-function-decl-args                //      -bfda
        --break-function-decl-args-end            //      -bfde
        --case-indentation2                       //      -clin
        --case-brace-indentation0                 //      -cbin
        --comment-delimiters-on-blank-lines       //      -cdb
//      --comment-indentation                     //      -cn
        --continuation-indentation2               //      -cin
//      --continue-at-parentheses                 //      -lp
        --cuddle-do-while                         //      -cdw
//      --cuddle-else                             //      -ce
//      --declaration-comment-column              //      -cdn
        --declaration-indentation2                //      -din
//      --dont-break-function-decl-args           //      -nbfda
//      --dont-break-function-decl-args-end       //      -nbfde
        --dont-break-procedure-type               //      -npsl
//      --dont-cuddle-do-while                    //      -ncdw
        --dont-cuddle-else                        //      -nce
//      --dont-format-comments                    //      -nfca
//      --dont-format-first-column-comments       //      -nfc1
        --dont-line-up-parentheses                //      -nlp
        --dont-left-justify-declarations          //      -ndj
        --dont-space-special-semicolon            //      -nss
        --dont-star-comments                      //      -nsc
        --else-endif-column1                      //      -cpn
        --format-all-comments                     //      -fca
        --format-first-column-comments            //      -fc1
//      --gnu-style                               //      -gnu
        --honour-newlines                         //      -hnl
//      --ignore-newlines                         //      -nhnl
        --ignore-profile                          //      -npro
```

```
      --indent-label1                            //     -il n
      --indent-level2                            //     -in
//      --k-and-r-style                          //       -kr
//      --leave-optional-blank-lines             //       -nsob
//      --leave-preprocessor-space               //       -lps
//      --left-justify-declarations              //       -dj
//      --line-comments-indentation              //       -dn
      --line-length80                            //       -ln
//      --linux-style                            //       -linux
      --no-blank-lines-after-commas              //     -nbc
//      --no-blank-lines-after-declarations      //       -nbad
//      --no-blank-lines-after-procedures        //       -nbap
//      --no-blank-lines-before-block-comments   //       -nbbb
//      --no-comment-delimiters-on-blank-lines   //       -ncdb
      --no-space-after-casts                     //     -ncs
//      --no-parameter-indentation               //       -nip
      --no-space-after-for                       //     -nsaf
      --no-space-after-function-call-names       //     -npcs
      --no-space-after-if                        //     -nsai
      --no-space-after-parentheses               //     -nprs
      --no-space-after-while                     //     -nsaw
      --no-tabs                                  //     -nut
//      --no-verbosity                           //       -nv
//      --original                               //       -orig
      --parameter-indentation2                   //     -ipn
      --paren-indentation0                       //     -pin
      --preserve-mtime                           //     -pmt
//      --preprocessor-indentation               //       -ppin
//      --procnames-start-lines                  //       -psl
//      --space-after-cast                       //       -cs
//      --space-after-for                        //       -saf
//      --space-after-if                         //       -sai
//      --space-after-parentheses                //       -prs
//      --space-after-procedure-calls            //       -pcs
//      --space-after-while                      //       -saw
//      --space-special-semicolon                //       -ss
//      --standard-output                        //       -st
//      --start-left-side-of-comments            //       -sc
//      --struct-brace-indentation               //       -sbin
      --swallow-optional-blank-lines             //     -sob
      --tab-size2                                //     -tsn
//      --use-tabs                               //       -ut
      --verbose                                  //     -v
```

● Note that you must specify additionally all type definitions in your sources with the `-T`–option.

● See the **`indent`**–manpage for further details on how to invoke the formatter.

The generated files still have some *problems* in not fulfilling the rigorously the **LIA** style recommendations:

- The closing brace of function declarations and definitions, i.e., after the parameter declaration, is not put into a new line and not aligned to the suggested column.
- There are introduced to much empty lines between declarations.

- The variables are not declared all in their own line.
- The space between `switch` and the following parenthesis is not removed.
- Some comments after code is broken incorrectly when they spread over more than one line which may lead to uncompilable code.
- The pointer `*` is not placed directly after the typename.
- Some strings are not aligned nicely when used as literal parameters. Especially, they are not broken up into nicely readable parts.
- Code embedded into commants is formated as comment, not as code.
- Mixed C–style and C++–style comments are not always dealt with correctly.