mCWEB, an Extension of CWEB for Teams

Markus Öllinger

March 3, 1997

Abstract

This article describes mCWEB—a descendant of the CWEB system of structured documentation by Donald E. Knuth and Silvio Levy—that adds some features that are indispensable when working in a team. mCWEB regards a project as a book consisting of several chapter files. By means of import and export commands, it automatically manages all relationships between the chapters of a book and to other books.

Interface Documentation is now part of the mCWEB file and can be extracted into a second TEX file. This allows to define well known interfaces between the individual parts of a project that will be implemented by different persons.

In [6] Donald E. Knuth introduced the term 'Literate Programming'. Knuth tought that programs should best be seen as works of literature that are meant to be read by human beings. The main difference to conventional methods is that a programmer is supposed to write the program as if he or she wants to present it to another programmer, which means that you should present the program in an order that is suitable to understand it easily.

Knuth has created two preprocessor tools which make it possible to combine documentation and code in one document, the WEB system of structured documentation [5]. It has been used to write programs of any size, from rather small examples [2, 3] to rather large ones like TEX [7] and METAFONT [8].

Later, Knuth's WEB has been ported to different languages and further extensions were made to the syntax of the language. A couple of other tools like noweb [12], nuweb [4] or FWEB [1] appeared. Silvio Levy created the C version of WEB called CWEB [10].

Although the CWEB system also makes sense for smaller programs, its strong points lie in the support of large scale software projects, where accurate documentation and careful design are basic requirements for success. But nowadays, most large projects are developed by software teams rather than one single programmer. This means, that the program will necessarily consist of more than one WEB source file, since one file can only be edited by one person at a time.

Unfortunately, CWEB does not support multiple source files at the moment. It requires one single source document, which makes it difficult to use for projects where more than one person is involved. Even for projects which are developed by a single person it may be desirable to split the source file up into more files.

There is a historical reason for this inconvenience. CWEB is a descendant of Knuth's WEB which was intended for PASCAL. PASCAL, unlike C, does not support linking of multiple files together and did indeed expect one single input file. However, the adaption of WEB to a new language like C does not only mean changing the grammar for pretty-printing, but you also have to consider the language specific properties.

One of the features of C are header files, which are used to insert shared data into different translation units. In release 3 of the CWEB system of structured documentation, the new `@(` command was introduced, which allows writing some sections into another file but the C output file. With this option, it is possible to create C header files.

```
@ @(foo.h@>=
  void bar(int);
```

13. ⟨foo.h 13⟩=
void *bar*(**int**);

outputs the code of this section to the file `foo.h`.

This header file can then be included with **#include** "foo.h", but if we are using `make`, the header file is rewritten every time we call CTANGLE thus causing a retranslation of the whole project and making `make` useless. Knuth's demonstrates how he uses `@(` in [9]. Still, the index only covers one single CWEB file and is unaware of all identifiers in other files.

Anyway, header files are concessions to the compiler because from a human point of view, there is no reason for packing some parts of a program in separate header files. The mCWEB [11] system described below has a couple of export and import commands which make header files for information interchange between translation units obsolete.

Furthermore, software teams usually break their projects into smaller, independent units and assign each programmer one of it. To coordinate the work between the individual members of the team, an interface specification is made which does not cover the implementation details but only how the different units interact with each other. This means that each unit is regarded as a black box with a well-defined interface. mCWEB is able to combine the interface documentation in the web code.

mCWEB's Book Concept

Conforming to the idea of creating works of literature, I decided to regard each executable or library as a *book* consisting of several chapters. Each chapter is a single file that can be edited independently from all other files. Thus, an old-style CWEB source file would only be a chapter of a mCWEB book. Chapters are meant to contain functions that have a high cohesion (e.g. they work on the same data structure).

Book files usually have the file extension `.prg`. For instance, an example book `foobar.prg` could look like the following:

```
\def\title{The FooBar Program}
\def\author{A. U. Thor}
\showtitle
\noindent
This is an example book.
\vfill
@c foo
@c bar
@m
#
# Here is the makefile
#
foobar :  $(CHAPTERS)
   $(CC) -o foobar $(CHAPTERS)
foo.o :  $(FOO)
bar.o :  $(BAR)
```

As we can see, the book imports two chapters `foo.w` and `bar.w`. Everything following the `@m` [⟨*makefile name*⟩] command goes to the makefile.

`mCWEAVE` processes all chapters given by the `@c` commands and copies all other lines of the book file up to the `@m` command or the end of file to the output file `foobar.tex`. This means that one can define TEX macros in the limbo part of the book that will be available in all chapters of the book.

In particular, one can write an introduction to the book after the `\showtitle` command like it is indicated in the above example. This introduction should contain information about what the program the book contains does. In addition, if it is necessary to have read other books in order to understand this one, please say so in the introduction so that reader can find out which book to start reading with if a project consists of multiple books.

The Chapter Files

Chapter files are ordinary `CWEB` files and therefore end in `.w`. Each chapter file usually starts with the TEX macro

`\chapter` *name-of-chapter*.

giving the name of the chapter (terminated by a '`.`') which will also be used in the header. Unlike in former `CWEB` files, the first section should *not* be a starred section because the `\chapter` macro already puts a chapter title line and adjusts the header. Use an ordinary `@␣` instead to introduce the first section.

Unlike `CWEB`, `mCWEB` now parses all included header files, so that it now knows about all datatypes defined there. In addition, datatypes appearing in the `CWEB` source are now recognized as such even if they are referenced before they are defined in the source file.

To translate a book to a TEX file, just call `mCWEAVE` as one did call `CWEAVE` in order to translate old-style `CWEB` files. Files with the recommended book file extension `.prg` are automatically treated as book files. If one has chosen another file extension, one must explicitly set the '`+m`' flag or `mCWEB` will be in `CWEB` compatibility mode.

Similarly, just pass the book file to `mCTANGLE` to convert it into a compileable C file. `mCTANGLE` will only translate those chapters that have changed. This means that chapters that have not been modified will not create new C files, so that the compiler won't have to retranslate them.

`mCWEAVE` always weaves all chapters of the book. For an input file `foobar.prg`, `mCWEAVE` outputs a file `foobar.tex` containing the implementation (as `CWEB` did) and—if there is an interface documentation in the book—a file `autodoc.tex`. Both files are plainTEX files and can be passed on to TEX to get printable DVI files.

Import and Export

As mentioned above, `mCWEB` now supports export and import commands which greatly simplify the maintenance of the relationship between the individual source files that make up a project. What we want to do is to automatically generate declarations for all parts of a chapter we want to make visible to others. For example, if one has a function *func* in chapter A one wants to be accessible from other chapters of the same book, one simply writes:

shared int *func*(**int** *x*)
{
...
}

In chapter B of the same book, one can write

#import chapter `"A"`

and chapter B will automatically have a prototype of chapter A's function *func* thus making it able to call it (as well as all other **shared** functions defined in chapter A). This does not only work for functions but for all C definitions like datatypes or variables.

Export Commands

Let's have a look in more detail what export commands are available. There are three export levels. There first level is indicated by the `@_global` qualifier. It makes a function visible in the whole chapter where it is defined, which means that—unlike in ordinary C—the function can be called before it is defined. This makes it easy to rearrange sections without caring about their interrelationships. It is generally a good idea to precede every C function by `@_global` which saves the work of creating all function prototypes by hand and putting them into annoying ⟨ Predeclaration of procedures ⟩ sections.

The next export level makes the function callable by another chapter of the same book. The name of this export command is `@_shared`, where `@_shared` implies `@_global` (i.e. all shared functions are automatically global).

Last but not least, one can export functions to another book. This can be done using `@_export` in front of the C definition. Exported functions are *not* automatically global nor shared so one might want to combine two export commands to make a function, say, 'exported and shared'.

Exporting to other books is often necessary if a project is made of more than one book. This is the case in Client-/Server Applications and projects that make use of libraries. Since libraries are books too, they are supposed to `@_export` all their interface functions to make them visible for users of the library.

The `mCWEB` system automatically creates the required header files (called shared and export files) for each chapter which consist of declarations for the exported stuff.

Import Commands

In order to import exported data, we can either import from another chapter of the same book, from a chapter of another book or we can simply import all exported stuff from all chapters of another book.

Import commands can have the optional keyword `transitively` which determines if these imported chapters are passed on transitively to whatever imports this chapter. If, for example, chapter B transitively imports chapter A and chapter C in turn imports chapter B, then chapter C automatically imports chapter A, too. However, if `transitively` was omitted, chapter A would not automatically be imported into chapter C.

The Improved Index

Since a `mCWEB` book consists of several chapters and can be even related to other books, we have to print a more complete index than `CWEB` did. `mCWEB` outputs an index at the end of each chapter and a final index at the end of the book.

At the end of each chapter one gets an index containing all identifiers defined in this chapter or imported from other chapters. Let's take a look at an excerpt of a chapter index:

> *AddHead*: $\underline{1}^{2}$, 19.
> *AddIcon*: $\underline{19}^{\dagger}$, 24, 30, 39.
> ...
> *compare_icons*: 20, $\underline{21}$.

In this index, *compare_icons* is a function that is defined in the current chapter and not exported. *AddIcon* is also defined in this chapter, but the † sign indicates, that this identifier is shared between chapters within the book. Identifiers exported to other books are marked with a ‡ sign.

Each imported identifier has a superscript number telling where it comes from. In our example, *AddHead* was defined in section 1 of another part of the project and is used in section 19 of the chapter the index belongs to.
At the end of the index we have a description of the superscript indices like:

> † shared within book
>
> **Referenced books:**
> ^{1} `lists`, Chapter 1
> ^{2} `lists`, Chapter 2
> ...

Thus, *AddHead* is defined in book `lists`, Chapter 2.

The final index consists of three parts. First come the shared identifiers, which means everything that is exchanged between the chapters of the book, followed by the exported identifiers (all names exported to other books). Last but not least comes the list of all identifiers imported from other books into the current one. Like in the chapter index, the entries in the final index have superscripts to indicate where they really come from.

Dependencies and Makefile

I presume that every software team uses `make` or a similar tool which helps to automatically keep a project up-to-date by only retranslating the files that have changed. This requires a `makefile` to give all the dependencies of the files.

These dependencies are not trivial if one has a large program consisting of several libraries where individual parts of the program **#include** many header files, some of them resulting from transitive dependencies.

Due to import and export commands, `mCTANGLE` knows about that interrelationship of its chapter files and can help the programmer creating the dependencies for the `makefile`. As we have seen, the book file can contain a `@m` command followed by an optional name of the makefile. All the lines following this command until EOF are copied to this makefile without change.

When `mCTANGLE` writes the makefile, it puts some useful constants at the very beginning of the file. For each chapter, there is a makefile constant with the name of the chapter which contains all files this chapter depends on.

Linking is simplified by the `CHAPTERS` constant which contains all object files that emerge from the book.

For a book `foobar.prg` consisting of two chapters named `foo.w` and `bar.w`, `mCTANGLE` defines `FOO= foo.c ...`, `BAR= bar.c ...` and `CHAPTERS= foo.o bar.o`.

The last makefile constant defined by `mCTANGLE` is `LIBRARIES` which contains all associated link libraries the book depends on.

Inserting Example Code

Sometimes, one might want to give an example to illustrate the use of a function or datatype. `CWEB` provides the `|...|` instruction in TeX text to set identifiers like C text, but it didn't work for C text that contains multiple lines. For this reason, `mCWEB` now knows the `@e` command which switches example mode on and off. One can use `@e` in the TeX text part of a section. Example code may only contain ordinary C code, no named sections.

Interface Documentation

In software teams, the individual members are not interested in the implementation details of those parts, which have been written by other members of the group. They rather would like to see each part reduced to an interface documentation, so they can see another user's chapter from outside without having to read the whole document.

Unfortunately, the `CWEB` system did not support interface documentation. Therefore, a so called *autodoc* section has been added to `mCWEB`. Autodoc sections may only appear in the TeX text of a section and are exported to a separate book called `autodoc.tex`.

The autodoc command has the following syntax:

`@a`{⟨ *class* ⟩} {⟨ *name* ⟩} {⟨ *TeX-text* ⟩}

Each autodoc entry has a name and belongs to an *autodoc class*. For each class *classname*, `mCWEAVE` creates a file *classname*`.adc` with all autodocs sorted by their name. Autodoc classes can be used to group the same kind of things. For example, for simple books, one may create a class `Functions` and a class `Datatypes`, thus having a chapter in the reference manual which describes all functions and one for the various datatypes. In a more complex project one might refine this classification. Of course, the autodoc book provides a table of contents containing the names of the autodoc classes and all autodoc entries within.

The idea of putting the interface documentation in the same file as the source file is obvious. If the interface is documented at the same place where the code is, it is easy to keep both up-to-date if changes are necessary. One doesn't have to search the corresponding places in two separate files.

Copy & Paste

Sometimes, one wants to have a part of the code in two different places. In this case, one would usually use named sections. But if you want to have it printed by TEX in both places, you will have to copy it manually. In addition, named sections only work for code and not for ordinary text since it is `mCTANGLE`'s job to replace the named sections by their real contents.

Copying and inserting by means of an editor is not only a nuisance but also dangerous because in case one of the copies gets changed, one will have to change all of them (which are sometimes hard to remember). So, every time one needs consistent copies of parts of the code, one can use `mCWEAVE`'s new mark/copy/paste commands.

Suppose, one has a `C` structure and also wants to have it in an autodoc section, something that really happens very often. Then one encloses the part of the code to copy in `@_mark "`⟨*name*⟩`"` and `@_copy`, thus assigning it the name ⟨*name*⟩. This code chunk can be inserted anywhere in the `mCWEB` source code using `@_paste "`⟨*name*⟩`"`.

The scope of the copy buffer commands is a chapter. Note that one can paste a copy buffer even before defining it.

Conclusions

Since `mCWEB` is based on the source code of `CWEB`, it is fully backwards compatible. Users of `CWEB` will have no problems switching to the new system. Furthermore, its' enhancements make it possible to split up a project into several translation units (called chapters) and executeables/libraries (called books). This makes it possible for software teams to write well documented programs in a literary style while still having multiple input files.

Making `CWEB` usable for teams was especially important because large scale projects always consist of multiple files and are generally harder to understand than small single file programs created by an individual programmer. Since a well documented literate program can greatly help keeping a project manageable and understandable, software teams will particularly like to take advantage of the literate programming paradigm.

References

[1] A. Avenarius and S. Oppermann. `FWEB`: A literate programming system for Fortran 8X. *ACM SIGPLAN Notices*, 25(1):52–58, Jan. 1990.

[2] J. Bentley. Programming pearls—literate programming. *Communications of the Association for Computing Machinery*, 29(5):364–369, May 1986.

[3] J. Bentley, D. E. Knuth, and D. McIlroy. Programming pearls—a literate program. *Communications of the Association for Computing Machinery*, 29(6):471–483, June 1986.

[4] P. Briggs. Nuweb, A simple literate programming tool. `cs.rice.edu:/public/preston`, Rice University, Houston, TX, 1993.

[5] D. E. Knuth. The `WEB` system of structured documentation. Stanford Computer Science Report CS980, Stanford University, Stanford, CA, Sept. 1983.

[6] D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, May 1984.

[7] D. E. Knuth. *TEX: The Program*, volume B of *Computers & Typesetting*. Addison-Wesley, Reading, MA, USA, 1986.

[8] D. E. Knuth. *METAFONT: The Program*, volume D of *Computers & Typesetting*. Addison-Wesley, Reading, MA, USA, 1986.

[9] D. E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing*. ACM Press, New York, NY 10036, USA, 1994. [From the publisher]: ... represents Knuth's final preparation for Volume 4 of *The Art of Computer Programming*. Through the use of about 30 examples, the book demonstrates the art of literate programming. Each example is a programmatic essay, a short story that can be read by human beings, as well as read and interpreted by machines. In these essays/programs, Knuth makes new contributions to the exposition of several important algorithms and data structures.

[10] D. E. Knuth and S. Levy. *The CWEB System of Structured Documentation, Version 3.0*. Addison-Wesley, Reading, MA, USA, 1993.

[11] M. Öllinger. `mCWEB`, an Extension of `CWEB` for Teams. Manual and software at `ist.tu-graz.ac.at` in `/pub/litprog/mcweb/mcweb.tgz`. Institute of Software Technology, TU-Graz, Oct. 1996.

[12] N. Ramsey. Literate-programming tools need not be complex. Report at `ftp.cs.princeton.edu` in `/reports/1991/351.ps.Z`. Software at `ftp.cs.princeton.edu` in `/pub/noweb.shar.Z` and at `bellcore.com` in `/pub/norman/noweb.shar.Z`. CS-TR-351-91, Department of Computer Science, Princeton University, Aug. 1992. Submitted to *IEEE Software*.