Literate-Programming Can Be Simple and Extensible

Norman Ramsey
Department of Computer Science, Princeton University
35 Olden Street, Princeton, New Jersey 08544

October 1993

Abstract

When it was introduced, literate programming meant WEB. Desire to use WEB with languages other than Pascal led to the implementation of many versions. WEB is complex, and the difficulty of using WEB creates an artificial barrier to experimentation with literate programming. noweb provides much of the functionality of WEB, with a fraction of the complexity. noweb is independent of the target programming language, and its formatter-dependent part is less than 40 lines. noweb is extensible, because it uses two representations of programs: one easily edited by authors and one easily manipulated by tools.

This paper explains how to use the noweb tools and gives examples of their use. It sketches the implementation of the tools and describes how new tools are added to the set. Because WEB and noweb overlap, but each does some things that the other cannot, this paper enumerates the differences.

Key words: literate programming, readability, programming environments

Introduction

When literate programming was introduced, it was synonymous with WEB, a tool for writing literate Pascal programs [6, Chapter 4]. The idea attracted attention; several examples of literate programs were published, and a special forum was created to discuss literate programming [1, 2, 6, 13]. WEB was adapted to programming languages other than Pascal [3, 7, 8, 10, 12]. With experience, many WEB users became dissatisfied [9]. Some found WEB

not worth the trouble, as did one author of the program appearing in Appendix C of Reference 11. Others built their own systems for literate programming. The literate-programming forum was dropped, on the grounds that literate programming had become the province of those who could build their own tools [14].

`WEB` programmers interleave source code and descriptive text in a single document. When using `WEB`, a programmer divides the source code into *modules*. Each module has a documentation part and a code part, and modules may be written in any order. The programmer is encouraged to choose an order that helps explain the program. The code parts are like macro definitions; they have names, and they contain both code and references to other modules. A `WEB` file represents a single program; `TANGLE` extracts that program from the `WEB` source. One special module has a code part with no name, and `TANGLE` expands the code part of that module to extract the program. `WEAVE` converts `WEB` source to TeX input, from which TeX can produce high-quality typeset documentation of the program.

`WEB` is a complex tool. In addition to enabling programmers to present pieces of a program in any order, it expands three kinds of macros, prettyprints code, evaluates some constant expressions, provides an integer representation for string literals, and implements a simple form of version control. The manual for the original version documents 27 "control sequences" [5]. The versions for languages other than Pascal offer slightly different functions and different sets of control sequences. Significant effort is required to make `WEB` usable with a new programming language, even when using a tool designed for that purpose [8].

`WEB`'s shortcomings make it difficult to explore the *idea* of literate programming; too much effort is required to master the *tool*. I designed a new tool that is both simple and independent of the target programming language. `noweb` is designed around one idea: writing named chunks of code in any order, with interleaved documentation. Like `WEB`, and like all literate-programming tools, it can be used to write a program in pieces and to present those pieces in an order that helps explain the program. `noweb`'s value lies in its simplicity, which shows that the idea of literate programming does not require the complexity of `WEB`.

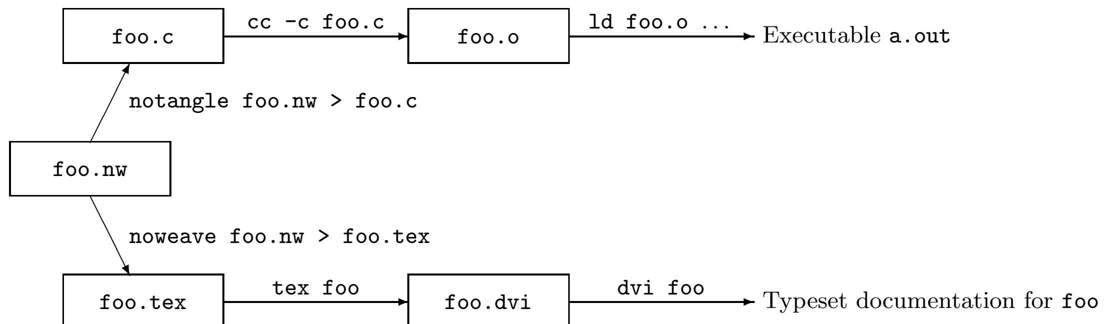

Figure 1: Using `noweb` to build code and documentation

`noweb`

A `noweb` file contains program source code interleaved with documentation. When `notangle` is given a `noweb` file, it writes the program on standard output. When `noweave` is given a `noweb` file, it reads the `noweb` source and produces, on standard output, TeX source for typeset documentation. Figure 1 shows how to use `notangle` and `noweave` to produce code and documentation for a C program contained in the `noweb` file `foo.nw`.

A `noweb` file is a sequence of *chunks*, which may appear in any order. A chunk may contain code or documentation. Documentation chunks begin with a line that starts with an at sign (`@`) followed by a space or newline. They have no names. Code chunks begin with

<<*chunk name*>>=

on a line by itself. The double left angle bracket (`<<`) must be in the first column. Chunks are terminated by the beginning of another chunk, or by end of file. If the first line in the file does not mark the beginning of a chunk, it is assumed to be the first line of a documentation chunk.

Documentation chunks contain text that is ignored by `notangle` and copied verbatim to standard output by `noweave` (except for quoted code). `noweave` can work with LaTeX, or it can use a TeX macro package, supplied with `noweb`, that defines commands like `\chapter` and `\section`.

Code chunks contain program source code and references to other code chunks. Several code chunks may have the same name; `notangle` concate-

nates their definitions to produce a single chunk, just as TANGLE does. Code chunk definitions are like macro definitions; notangle extracts a program by expanding one chunk (by default the chunk named <<*>>). The definition of that chunk contains references to other chunks, which are themselves expanded, and so on. notangle's output is readable; it preserves the indentation of expanded chunks with respect to the chunks in which they appear.

Code may be quoted within documentation chunks by placing double square brackets around it ([[...]]). These double square brackets are ignored by notangle, but they are used by noweave to give code special typographic treatment.

If double left and right angle brackets are not paired, they are treated as literal "<<" and ">>". Users can force any such brackets, even paired brackets, to be treated as literal by preceding the brackets by an at sign (e.g. "@<<").

Figure 2 shows a fragment of a noweb program that computes prime numbers. The program is derived from the example used in Reference 6, Chapter 4, and Figure 2 should be compared with Figure 2b of that paper. Figure 3 shows the program after processing by noweave and LaTeX. Figure 4 shows the beginning of the program as extracted by notangle. A complete example program accompanies this paper.

Using noweb

Experimenting with noweb is easy. noweb has little syntax: definition and use of code chunks, marking of documentation chunks, quoting of code, and quoting of brackets. noweb can be used with any programming language, and its manual fits on two pages.

On a large project, it is essential that compilers and other tools be able to refer to locations in the noweb source, even though they work with notangle's output [9]. Giving notangle the -L option makes it emit pragmas that inform compilers of the placement of lines in the noweb source. It also preserves the columns in which tokens appear. If notangle is not given the -L option, it respects the indentation of its input, making its output easy to read. Large programs may also benefit from cross-reference information. If given the -x option, noweave uses LaTeX to show on what pages each chunk is defined and used.

WEB files map one to one with to both programs and documents. The mapping of noweb files to programs is many to many; the mapping of files

```
@ This program has no input, because we want to keep it
simple.  The result of the program will be to produce a
list of the first thousand prime numbers, and this list
will appear on the [[output]] file.

Since there is no input, we declare the value [[m = 1000]]
as a compile-time constant.  The program itself is capable
of generating the first [[m]] prime numbers for any
positive [[m]], as long as the computer's finite
limitations are not exceeded.
<<program to print the first thousand prime numbers>>=
program print_primes(output);
  const m = 1000;
        <<other constants of the program>>
  var <<variables of the program>>
    begin <<print the first [[m]] prime numbers>>
    end.
```

Figure 2: Sample `noweb` input, from prime number program

This program has no input, because we want to keep it simple. The result of the program will be to produce a list of the first thousand prime numbers, and this list will appear on the `output` file.

Since there is no input, we declare the value `m = 1000` as a compile-time constant. The program itself is capable of generating the first `m` prime numbers for any positive `m`, as long as the computer's finite limitations are not exceeded.

⟨*program to print the first thousand prime numbers*⟩≡
```
  program print_primes(output);
    const m = 1000;
          ⟨other constants of the program⟩
    var ⟨variables of the program⟩
      begin ⟨print the first m prime numbers⟩
      end.
```

Figure 3: Output produced by `noweave` and LaTeX from Figure 2

```
program print_primes(output);
  const m = 1000;
        rr = 50;
        cc = 4;
        ww = 10;
        ord_max = 30;  { p_ord_max squared must exceed p_m }
  var p: array [1..m] of integer;
           { the first m prime numbers, in increasing order }
      page_number: integer;
    ⋮
```

Figure 4: Part of primes program as written by `notangle`

to documents is many to one. Source files are combined by listing their names on `notangle`'s or `noweave`'s command line. Many programs may be extracted from one source by specifying the names of different root chunks, using `notangle`'s `-R` command-line option.

The simplest example of a one-to-many mapping of programs is that of putting C header and program in a single `noweb` file. The header comes from the root chunk ⟨*header*⟩, and the program from the default root chunk, ⟨*⟩. The following rules for `make` automate the process:[1]

```
foo.c: foo.nw
        notangle -L foo.nw > foo.c
foo.h: foo.nw
        notangle -Rheader foo.nw > xfoo.h
        -cmp -s xfoo.h foo.h || cp xfoo.h foo.h
```

A more interesting example is using `noweb` to interleave different languages in one source file. I wrote an `awk` script that read a machine description and emitted a disassembler for that machine, and I used `noweb` to combine the script and description in a single file, so I could place each part of the input next to the code that processed that input. The machine description was in the root chunk ⟨*opcodes table*⟩, and the `awk` script in the default root chunk. The processing steps were:

```
notangle opcodes.nw > opcodes.awk
notangle -R'opcode table' opcodes.nw |
awk -f opcodes.awk > disassem.sml
```

Many-to-one mapping of source to program can be used to obtain effects similar to those of Ada or Modula-3 generics. Figure 5 shows generic C code that supports lists. The code can be "instantiated" by combining it with another `noweb` file. `pair_list.nw`, shown in Figure 6, specifies lists of integer pairs. The two are combined by applying `notangle` to them both:

```
notangle pair_list.nw generic_list.nw > pair_list.c
```

`noweb` has no parameter mechanism, so the "generic" code must refer to a fixed set of symbols, and it cannot be checked for errors except by compiling `pair_list.c`. These restrictions make `noweb` a poor approximation to real generics, but useful nevertheless.

[1] Using `cmp` avoids touching the header file when its contents haven't changed. This trick is explained on pages 265–266 of Reference 4.

This list code supports circularly-linked lists represented by a pointer to the last element. It is intended to be combined with other noweb code that defines ⟨*fields of a list element*⟩ (the fields found in an element of a list) and that uses ⟨*list declarations*⟩ and ⟨*list definitions*⟩.

⟨*list declarations*⟩≡

```
typedef struct list {
  ⟨fields of a list element⟩
  struct list *_link;
} *List;

extern List singleton(void);    /* singleton list, uninitialized fields */
extern List append(List, List); /* destructively append two lists */
#define last(l)   (l)
#define head(l)   ((l) ? (l)->next : 0)
#define forlist(p,l) for (p=head(l); p; p=(p==last(l) ? 0 : p->next))
```

⟨*list definitions*⟩≡

```
List append (List left, List right) {
    List temp;
    if (left == 0)  return right;
    if (right == 0) return left;
    temp = left->_link; left->_link = right->_link; right->_link = temp;
    return right;
}
  ⋮
```

Figure 5: Generic code for implementing lists in C

```
⟨*⟩≡
  ⟨list declarations⟩
  ⟨list definitions⟩
```

```
⟨fields of a list element⟩≡
  int x;
  int y;
```

Figure 6: Program to instantiate lists of integer pairs

I have used `noweb` for small programs written in various languages, including C, Icon, `awk`, and Modula-3. Larger projects have included a code generator for Standard ML of New Jersey (written in Standard ML) and a multi-architecture debugger, written in Modula-3, C, and assembly language. A colleague used `noweb` to write an experimental file system in C++. The sizes of these programs are

Program	Documentation lines	Total lines
`markup` and `nt`	400	1,200
ML code generator	900	2,600
Debugger	1,400	11,000
File system	4,400	27,000

Representation of `noweb` files

The `noweb` syntax is easy to read, write, and edit, but it is not easily manipulated by programs. To make it easy to extend `noweb`, I have written `markup`, which converts `noweb` source to a representation that is easily manipulated by commonly used Unix tools like `sed` and `awk`. In this representation, every line begins with `@` and a key word. The possibilities are:

`@begin` *kind n*	Start a chunk
`@end` *kind n*	End a chunk
`@text` *string*	*string* appeared in a chunk
`@nl`	A newline
`@defn` *name*	The code chunk named *name* is being defined
`@use` *name*	A reference to code chunk named *name*
`@quote`	Start of quoted code in a documentation chunk
`@endquote`	End of quoted code in a documentation chunk
`@file` *filename*	Name of the file from which the chunks came
`@index defn` *ident*	The current chunk contains a definition of *ident*
`@index use` *ident*	The current chunk contains a use of *ident*
`@index nl` *ident*	A newline that is part of markup, not part of the chunk
`@literal` *text*	`noweave` copies *text* to output

`markup` numbers each chunk, starting at 0. It also recognizes and undoes the escape sequence for double brackets, e.g. converting "`@<<`" to "`<<`". `markup`'s output represents a sequence of files. Each file is represented by a "`@file` *filename*" line, followed by a sequence of chunks.

The representation of a documentation chunk is

`@begin docs` *n*	where *n* is the chunk number.
docline	repeated an arbitrary number of times.
`@end docs` *n*	

where *docline* may be `@text`, `@nl`, `@quote`, `@endquote`, or `@index`. Every `@nl` corresponds to a newline in the original file. `markup` guarantees that quotes are balanced and not nested.

The representation of a code chunk is

`@begin code` *n*	where *n* is the chunk number.
`@defn` *name*	name of this chunk.
`@nl`	The newline following `<<`*name*`>>=` in the original file
codeline	repeated an arbitrary number of times.
`@end code` *n*	

where *codeline* may be `@text`, `@nl`, `@use`, or `@index`.

The `noweb` tools are implemented by piping the output of `markup` to other programs. `notangle` is a Unix shell script that builds a pipeline between `markup` and `nt`, which reads and expands definitions of code chunks. `noweave` pipes the output of `markup` to a 24-line `awk` script that inserts appropriate TeX or LaTeX formatting commands.

Having a format easily read by programs makes noweb extensible; one can manipulate literate programs using Unix shell scripts and filters. To be able to share programs with colleagues who don't enjoy literate programming, I modified notangle by adding to its pipeline a stage that places each line of documentation in a comment and moves it to the succeeding code chunk. The resulting script, nountangle, transforms a literate program into a traditional commented program, without loss of information and with only a modest penalty in readability. Figure 7 shows the results of applying nountangle to the prime-number program shown in Figure 2. noweave's cross-reference generation is also implemented as an extension; the output of markup is piped through an awk script that uses @literal to insert LaTeX cross-reference commands. Another simple tool finds all the roots in a noweb file, making it easy to find definitions where chunk names have been misspelled.

Comparing WEB and noweb

Unlike WEB, noweb is independent of the target programming language. WEB tools can be generated for many programming languages, but those languages must be lexically similar to C. For example, WEB can't handle the awk regular-expression notation "/.../"; every such expression must quoted using WEB's "verbatim" control sequence. The effort required to generate WEB tools is significant; the prospective user must write a specification of several hundred lines.

Being independent of the target programming language makes noweb simpler, but it also means that noweb can do less. Most of the differences between WEB and noweb arise because WEB has language-dependent features that are not present in noweb. These features include prettyprinting, typesetting comments using TeX, generating an index of identifiers, expanding macros, evaluating constant expressions, and converting string literals to indices into a "string pool." Among these features, noweb users are most likely to miss prettyprinting and the index of identifiers.

Some differences arise because WEB and noweb implement similar features differently. WEB's original TANGLE removed white space and folded lines to fill each line with tokens, making its output unreadable [6, Chapter 4, Figure 3]. Later adaptations preserved line breaks but removed other white space. By default, notangle preserves whitespace and maintains indentation when

```
{ This program has no input, because we want to keep it    }
{ simple.  The result of the program will be to produce a  }
{ list of the first thousand prime numbers, and this list  }
{ will appear on the [[output]] file.                      }
  ⋮

{ <program to print the first thousand prime numbers>=     }
program print_primes(output);
  const m = 1000;
        { \section-The output phase-                               }
        {                                                          }
        { <other constants of the program>=                        }
        rr = 50;
        cc = 4;
        ww = 10;
        { <other constants of the program>=                        }
        ord_max = 30;  { p_ord_max squared must exceed p_m }
  var { How should table [[p]] be represented? Two possibilities }
      { suggest themselves: We could construct a sufficiently    }
  ⋮
```

Figure 7: Output produced by `nountangle` from Figure 2

expanding chunks. It can therefore be used with languages like Miranda and Haskell, in which indentation is significant. TANGLE cannot.

WEB's WEAVE assigns a number to each chunk, and its cross-reference information refers to chunk numbers, not page numbers. noweb uses LaTeX to emit cross-reference information that refers to page numbers. Anyone who has read a large literate program will appreciate the difference.

WEB works poorly with LaTeX; LaTeX constructs cannot be used in WEB source, and getting WEAVE output to work in LaTeX documents requires tedious adjustments by hand. noweb works with both plain TeX and LaTeX. Both WEAVE and noweave depend on the text formatter in two ways: the source of the program itself, and the supporting macros. WEAVE's source (written using WEB for C) is several thousand lines long, and the formatting code is not isolated. noweave's source is a 57-line shell script, and only 31 of those lines have to do with formatting. Both WEAVE and noweave use about 200 lines of supporting macros for plain TeX. noweb uses another 80 lines to support LaTeX, most of which is used to eliminate duplicate page numbers in cross-reference lists.

noweb has two features that weren't in the original WEB, but that appeared in some of WEB's later adaptations. They are the ability to inform the compiler of the original locations of source lines and the ability to extract more than one program from a single source file.

Reviewers have had many expectations of literate-programming tools [13, 14]. The most important is *verisimilitude*: a single input should produce both compilable program and publishable document, warranting the correctness of the document. Others include flexible order of elaboration, ability to develop program and documentation concurrently in one place, cross-references, and indexing. WEB satisfies all these expectations, and noweb satisfies all but one (it does not provide automatic indexing).

Discussion

WEB takes the monolithic approach to literate programming—it does everything. noweb's approach is to compose simple tools that manipulate files in the noweb format. Existing Unix tools provide some of the WEB features that aren't found in noweb. Unix supplies two macro processors: the C preprocessor and the m4 macro processor. xstr extracts string literals. patch provides a form of version control similar to WEB's change files. Few of WEB's remaining features will be missed; for example, many compilers evaluate

constant expressions at compile time. Experience with `WEB` has suggested that prettyprinting may be more trouble than it is worth, and that the index of identifiers, while useful, is not a necessity [9].

Three things distinguish `noweb` from previous work. `noweb` takes as simple as possible a view of literate programming and the tools needed to implement it. Instead of relying on a generator or re-implementation to support different programming languages, `noweb` is independent of the target programming language. `noweave`'s dependence on its typesetter is small and isolated, instead of being distributed throughout a large implementation.

Experimenting with `noweb` is easy because the tools are simple and they work with any language. If the experiment is unsatisfying, it is easy to abandon, because `notangle`'s output, unlike `TANGLE`'s, is readable. `noweb` is simpler than `WEB` and is easier to use and understand, but it does less. I argue, however, that the benefit of `WEB`'s extra features is outweighed by cost of the extra complexity, making `noweb` better for writing literate programs.

`noweb` can be obtained by anonymous `ftp` from `princeton.edu`, in file `pub/noweb.shar.Z`.

Acknowledgements

Mark Weiser's invaluable encouragement provided the impetus for me to write this paper, which I did while visiting the Computer Science Laboratory of the Xerox Palo Alto Research Center. Comments from David Hanson and from the anonymous referees stimulated me to improve the paper. The development of `noweb` was supported by a Fannie and John Hertz Foundation Fellowship.

References

[1] P. J. Denning. Announcing literate programming. *Communications of the ACM*, 30(7):593, July 1987.

[2] D. Gries and J. Bentley. Programming pearls: Abstract data types. *Communications of the ACM*, 30(4):284–290, April 1987.

[3] K. Guntermann and J. Schrod. `WEB` adapted to C. *TUGboat*, 7(3):134–137, October 1986.

[4] B. W. Kernighan and R. Pike. *The UNIX Programming Environment*. Prentice-Hall, 1984.

[5] D. E. Knuth. The WEB system of structured documentation. Technical Report 980, Stanford Computer Science, Stanford, California, September 1983.

[6] D. E. Knuth. *Literate Programming*, volume 27 of *Center for the Study of Language and Information Lecture Notes*. Leland Stanford Junior University, Stanford, California, 1992.

[7] S. Levy. WEB adapted to C, another approach. *TUGBoat*, 8(1):12–13, 1987.

[8] N. Ramsey. Literate programming: Weaving a language-independent WEB. *Communications of the ACM*, 32(9):1051–1055, September 1989.

[9] N. Ramsey and C. Marceau. Literate programming on a team project. *Software—Practice & Experience*, 21(7):677–683, July 1991.

[10] W. Sewell. How to MANGLE your software: the WEB system for Modula-2. *TUGboat*, 8(2):118–128, July 1987.

[11] W. Sewell. *Weaving a Program: Literate Programming in* WEB. Van Nostrand Reinhold, New York, 1989.

[12] H. Thimbleby. Experiences of 'literate programming' using cweb (a variant of Knuth's WEB). *Computer Journal*, 29(3):201–211, 1986.

[13] H. Thimbleby. A review of Donald C. Lindsay's text file difference utility, *diff*. *Communications of the ACM*, 32(6):752–755, June 1989.

[14] C. J. Van Wyk. Literate programming: An assessment. *Communications of the ACM*, 33(3):361–365, March 1990.

An example of `noweb`

The following short program illustrates the use of `noweb`, a low-tech tool for literate programming. The purpose of the program is to provide a basis for comparing `WEB` and `noweb`, so I have used a program that has been published before; the text, code, and presentation are taken from [6, Chapter 12]. The notable differences are:

- When displaying source code, `noweb` uses different typography. In particular, `WEB` makes good use of multiple fonts and the ablity to typeset mathematics, and it may use mathematical symbols in place of C symbols (e.g. "$\wedge$" for "`&&`"). `noweb` uses a single fixed-width font for code.
- `noweb` can work with LaTeX, and I have used LaTeX in this example.
- `noweb` has no numbered "sections." When numbers are needed for cross-referencing, `noweb` uses page numbers.
- `noweb` has no special support for macros. In the sample program, I have used a "⟨*Definitions*⟩" chunk to hold macro definitions.
- `noweb` does not produce an index of identifiers. Because it treats the program as text, not as C code, it cannot distinguish identifiers from other parts of the program.
- The `CWEB` version of this program has semicolons following most uses of ⟨···⟩. `WEB` needs the semicolon or its equivalent to make its prettyprinting come out right. Because it does not attempt prettyprinting, `noweb` needs no semicolons.

Counting words

This example, based on a program by Klaus Guntermann and Joachim Schrod [3] and a program by Silvio Levy and D. E. Knuth [6, Chapter 12], presents the "word count" program from UNIX, rewritten in `noweb` to demonstrate literate programming using `noweb`. The level of detail in this document is intentionally high, for didactic purposes; many of the things spelled out here don't need to be explained in other programs.

The purpose of `wc` is to count lines, words, and/or characters in a list of files. The number of lines in a file is the number of newline characters

it contains. The number of characters is the file length in bytes. A "word" is a maximal sequence of consecutive characters other than newline, space, or tab, containing at least one visible ASCII code. (We assume that the standard ASCII code is in use.)

Most literate C programs share a common structure. It's probably a good idea to state the overall structure explicitly at the outset, even though the various parts could all be introduced in chunks named ⟨*⟩ if we wanted to add them piecemeal.

Here, then, is an overview of the file `wc.c` that is defined by the `noweb` program `wc.nw`:

17a ⟨* 17a⟩≡
```
⟨Header files to include 17b⟩
⟨Definitions 17c⟩
⟨Global variables 18a⟩
⟨Functions 24⟩
⟨The main program 18b⟩
```
Root chunk (not used in this document).

We must include the standard I/O definitions, since we want to send formatted output to `stdout` and `stderr`.

17b ⟨*Header files to include* 17b⟩≡
```
#include <stdio.h>
```
This code is used on page 17a.

The `status` variable will tell the operating system if the run was successful or not, and `prog_name` is used in case there's an error message to be printed.

17c ⟨*Definitions* 17c⟩≡
```
#define OK               0      /* status code for successful run */
#define usage_error      1      /* status code for improper syntax */
#define cannot_open_file 2      /* status code for file access error */
```
Defines:
- `OK`, used on page 18a.
- `cannot_open_file`, used on page 20d.
- `usage_error`, used on page 24.

Uses `status` 18a.
This definition is continued on pages 20c, 21b, and 23d.
This code is used on page 17a.

18a ⟨*Global variables* 18a⟩≡

```
int status = OK;                /* exit status of command, initially OK */
char *prog_name;                /* who we are */
```

Defines:
 prog_name, used on pages 18b, 20d, and 24.
 status, used on pages 17c, 18b, 20d, and 24.
Uses OK 17c.
This definition is continued on page 22a.
This code is used on page 17a.

Now we come to the general layout of the main function.

18b ⟨*The main program* 18b⟩≡

```
main(argc, argv)
  int argc;          /* the number of arguments on the UNIX command line */
  char **argv;       /* the arguments themselves, an array of strings */
{
  ⟨Variables local to main 19a⟩
  prog_name = argv[0];
  ⟨Set up option selection 19b⟩
  ⟨Process all the files 20a⟩
  ⟨Print the grand totals if there were multiple files 23c⟩
  exit(status);
}
```

Defines:
 argc, used on pages 19b and 20a.
 argv, used on pages 19b, 20d, and 23a.
 main, never used.
Uses prog_name 18a and status 18a.
This code is used on page 17a.

If the first argument begins with a '-', the user is choosing the desired counts and specifying the order in which they should be displayed. Each selection is given by the initial character (lines, words, or characters). For example, '-cl' would cause just the number of characters and the number of lines to be printed, in that order.

We do not process this string now; we simply remember where it is. It will be used to control the formatting at output time.

19a ⟨*Variables local to* main 19a⟩≡

```
int file_count;   /* how many files there are */
char *which;      /* which counts to print */
```

Defines:
  file_count, used on pages 19b, 20d, and 23.
  which, used on pages 19b, 23, and 24.
This definition is continued on pages 20b and 21c.
This code is used on page 18b.

19b ⟨*Set up option selection* 19b⟩≡

```
which = "lwc";      /* if no option is given, print all three values */
if (argc > 1 && *argv[1] == '-') {
  which = argv[1] + 1;
  argc--;
  argv++;
}
file_count = argc - 1;
```

Uses argc 18b, argv 18b, file_count 19a, and which 19a.
This code is used on page 18b.

Now we scan the remaining arguments and try to open a file, if possible. The file is processed and its statistics are given. We use a `do ... while` loop because we should read from the standard input if no file name is given.

20a ⟨*Process all the files* 20a⟩≡

```
argc--;
do {
  ⟨If a file is given, try to open *(++argv); continue if unsuccessful 20d⟩
  ⟨Initialize pointers and counters 21d⟩
  ⟨Scan file 22b⟩
  ⟨Write statistics for file 23a⟩
  ⟨Close file 21a⟩
  ⟨Update grand totals 23b⟩    /* even if there is only one file */
} while (--argc > 0);
```

Uses `argc` 18b.
This code is used on page 18b.

Here's the code to open the file. A special trick allows us to handle input from `stdin` when no name is given. Recall that the file descriptor to `stdin` is 0; that's what we use as the default initial value.

20b ⟨*Variables local to* `main` 19a⟩+≡

```
int fd = 0;              /* file descriptor, initialized to stdin */
```

Defines:
`fd`, used on pages 20–22.

20c ⟨*Definitions* 17c⟩+≡

```
#define READ_ONLY 0      /* read access code for system open routine */
```

Defines:
`READ_ONLY`, used on page 20d.

20d ⟨*If a file is given, try to open* `*(++argv)`*;* `continue` *if unsuccessful* 20d⟩≡

```
if (file_count > 0 && (fd = open(*(++argv), READ_ONLY)) < 0) {
  fprintf(stderr, "%s: cannot open file %s\n", prog_name, *argv);
  status |= cannot_open_file;
  file_count--;
  continue;
}
```

Uses `READ_ONLY` 20c, `argv` 18b, `cannot_open_file` 17c, `fd` 20b, `file_count` 19a, `prog_name` 18a, and `status` 18a.
This code is used on page 20a.

21a ⟨*Close file* 21a⟩≡

```
close(fd);
```

Uses `fd` 20b.
This code is used on page 20a.

We will do some homemade buffering in order to speed things up: Characters will be read into the `buffer` array before we process them. To do this we set up appropriate pointers and counters.

21b ⟨*Definitions* 17c⟩+≡

```
#define buf_size BUFSIZ    /* stdio.h's BUFSIZ is chosen for efficiency */
```

Defines:
`buf_size`, used on pages 21c and 22c.

21c ⟨*Variables local to* `main` 19a⟩+≡

```
char buffer[buf_size];    /* we read the input into this array */
register char *ptr;       /* the first unprocessed character in buffer */
register char *buf_end;   /* the first unused position in buffer */
register int c;           /* current character, or number of characters just rea
int in_word;              /* are we within a word? */
long word_count, line_count, char_count;
                 /* number of words, lines, and characters found in the file so fa
```

Defines:
`buf_end`, used on pages 21d and 22c.
`buffer`, used on pages 21d and 22c.
`char_count`, used on pages 21–24.
`in_word`, used on pages 21d and 22b.
`line_count`, used on pages 21–24.
`ptr`, used on pages 21 and 22.
`word_count`, used on pages 21–24.
Uses `buf_size` 21b.

21d ⟨*Initialize pointers and counters* 21d⟩≡

```
ptr = buf_end = buffer;
line_count = word_count = char_count = 0;
in_word = 0;
```

Uses `buf_end` 21c, `buffer` 21c, `char_count` 21c, `in_word` 21c, `line_count` 21c, `ptr` 21c, and `word_count` 21c.
This code is used on page 20a.

The grand totals must be initialized to zero at the beginning of the program. If we made these variables local to `main`, we would have to do this initialization explicitly; however, C's globals are automatically zeroed. (Or rather, "statically zeroed.") (Get it?)

22a ⟨*Global variables* 18a⟩+≡

```
long tot_word_count, tot_line_count, tot_char_count;
                                  /* total number of words, lines, and chars */
```

The present chunk, which does the counting that is `wc`'s *raison d'être*, was actually one of the simplest to write. We look at each character and change state if it begins or ends a word.

22b ⟨*Scan file* 22b⟩≡

```
while (1) {
  ⟨Fill buffer if it is empty; break at end of file 22c⟩
  c = *ptr++;
  if (c > ' ' && c < 0177) {    /* visible ASCII codes */
    if (!in_word) {
      word_count++;
      in_word = 1;
    }
    continue;
  }
  if (c == '\n') line_count++;
  else if (c != ' ' && c != '\t') continue;
  in_word = 0;  /* c is newline, space, or tab */
}
```

Uses `in_word` 21c, `line_count` 21c, `ptr` 21c, and `word_count` 21c.
This code is used on page 20a.

Buffered I/O allows us to count the number of characters almost for free.

22c ⟨*Fill* `buffer` *if it is empty;* `break` *at end of file* 22c⟩≡

```
if (ptr >= buf_end) {
  ptr = buffer;
  c = read(fd, ptr, buf_size);
  if (c <= 0) break;
  char_count += c;
  buf_end = buffer + c;
}
```

Uses `buf_end` 21c, `buffer` 21c, `buf_size` 21b, `char_count` 21c, `fd` 20b, and `ptr` 21c.
This code is used on page 22b.

It's convenient to output the statistics by defining a new function wc_print; then the same function can be used for the totals. Additionally we must decide here if we know the name of the file we have processed or if it was just stdin.

23a ⟨*Write statistics for file* 23a⟩≡

```
wc_print(which, char_count, word_count, line_count);
if (file_count) printf(" %s\n", *argv);   /* not stdin */
else printf("\n");                        /* stdin */
```

Uses argv 18b, char_count 21c, file_count 19a, line_count 21c, wc_print 24, which 19a, and word_count 21c.
This code is used on page 20a.

23b ⟨*Update grand totals* 23b⟩≡

```
tot_line_count += line_count;
tot_word_count += word_count;
tot_char_count += char_count;
```

Uses char_count 21c, line_count 21c, and word_count 21c.
This code is used on page 20a.

We might as well improve a bit on UNIX's wc by displaying the number of files too.

23c ⟨*Print the grand totals if there were multiple files* 23c⟩≡

```
if (file_count > 1) {
  wc_print(which, tot_char_count, tot_word_count, tot_line_count);
  printf(" total in %d files\n", file_count);
}
```

Uses file_count 19a, wc_print 24, and which 19a.
This code is used on page 18b.

Here now is the function that prints the values according to the specified options. The calling routine is supposed to supply a newline. If an invalid option character is found we inform the user about proper usage of the command. Counts are printed in 8-digit fields so that they will line up in columns.

23d ⟨*Definitions* 17c⟩+≡

```
#define print_count(n) printf("%8ld", n)
```

Defines:
print_count, used on page 24.

24 ⟨*Functions* 24⟩≡

```
wc_print(which, char_count, word_count, line_count)
  char *which;                              /* which counts to print */
  long char_count, word_count, line_count;  /* given totals */
{
  while (*which)
    switch (*which++) {
      case 'l': print_count(line_count);
        break;
      case 'w': print_count(word_count);
        break;
      case 'c': print_count(char_count);
        break;
      default:
        if ((status & usage_error) == 0) {
          fprintf(stderr, "\nUsage: %s [-lwc] [filename ...]\n", prog_name);
          status |= usage_error;
        }
    }
}
```

Defines:
`wc_print`, used on page 23.

Uses `char_count` 21c, `line_count` 21c, `print_count` 23d, `prog_name` 18a, `status` 18a, `usage_error` 17c, `which` 19a, and `word_count` 21c.

This code is used on page 17a.

Incidentally, a test of this program against the system `wc` command on a SPARCstation showed that the "official" `wc` was slightly slower. Furthermore, although that `wc` gave an appropriate error message for the options '`-abc`', it made no complaints about the options '`-labc`'! Dare we suggest that the system routine might have been better if its programmer had used a more literate approach?

List of code chunks

⟨* 17a⟩
⟨*Close file* 21a⟩
⟨*Definitions* 17c⟩
⟨*Fill* `buffer` *if it is empty;* `break` *at end of file* 22c⟩
⟨*Functions* 24⟩
⟨*Global variables* 18a⟩
⟨*Header files to include* 17b⟩
⟨*If a file is given, try to open* `*(++argv)`*;* `continue` *if unsuccessful* 20d⟩
⟨*Initialize pointers and counters* 21d⟩
⟨*Print the grand totals if there were multiple files* 23c⟩
⟨*Process all the files* 20a⟩
⟨*Scan file* 22b⟩
⟨*Set up option selection* 19b⟩
⟨*The main program* 18b⟩
⟨*Update grand totals* 23b⟩
⟨*Variables local to* `main` 19a⟩
⟨*Write statistics for file* 23a⟩

Index

`OK`: 17c, 18a
`READ_ONLY`: 20c, 20d
`argc`: 18b, 19b, 20a
`argv`: 18b, 19b, 20d, 23a
`buf_end`: 21c, 21d, 22c
`buffer`: 21c, 21d, 22c
`buf_size`: 21b, 21c, 22c
`cannot_open_file`: 17c, 20d
`char_count`: 21c, 21d, 22c, 23a, 23b, 24
`fd`: 20b, 20d, 21a, 22c
`file_count`: 19a, 19b, 20d, 23a, 23c
`in_word`: 21c, 21d, 22b
`line_count`: 21c, 21d, 22b, 23a, 23b, 24
`main`: 18b
`print_count`: 23d, 24
`prog_name`: 18a, 18b, 20d, 24
`ptr`: 21c, 21d, 22b, 22c
`status`: 17c, 18a, 18b, 20d, 24
`usage_error`: 17c, 24
`wc_print`: 23a, 23c, 24
`which`: 19a, 19b, 23a, 23c, 24
`word_count`: 21c, 21d, 22b, 23a, 23b, 24