GNU cflow

version 1.6, 11 February 2019

Sergey Poznyakoff.

Published by the Free Software Foundation, 51 Franklin Street, Fifth Floor Boston, MA 02110-1301, USA

Short Contents

Table of Contents

1 Introduction to cflow

The `cflow` utility analyzes a collection of source files written in `C` programming language and outputs a graph charting dependencies between various functions.

The program is able to produce two kind of graphs: direct and reverse. *Direct graph* begins with the main function (`main`), and displays recursively all functions called by it. In contrast, *reverse graph* is a set of subgraphs, charting for each function its callers, in the reverse order. Due to their tree-like appearance, graphs can also be called *trees*.

In addition to these two output modes, `cflow` is able to produce a *cross-reference* listing of all the symbols encountered in the input files.

The utility also provides a detailed control over symbols that will appear in its output, allowing to omit those that are of no interest to the user. The exact appearance of the output graphs is also configurable.

2 Simple Ways to Analyze Programs with `cflow`.

Let's begin our acquaintance with the GNU `cflow` utility with an example. Suppose you have a simple implementation of `whoami` command and you wish to obtain a graph of function dependencies. Here is the program:

```
/* whoami.c - a simple implementation of whoami utility */
#include <pwd.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>

int
who_am_i (void)
{
  struct passwd *pw;
  char *user = NULL;

  pw = getpwuid (geteuid ());
  if (pw)
    user = pw->pw_name;
  else if ((user = getenv ("USER")) == NULL)
    {
      fprintf (stderr, "I don't know!\n");
      return 1;
    }
  printf ("%s\n", user);
  return 0;
}

int
main (int argc, char **argv)
{
  if (argc > 1)
    {
      fprintf (stderr, "usage: whoami\n");
      return 1;
    }
  return who_am_i ();
}
```

Running `cflow` produces the following output:

```
$ cflow whoami.c
main() <int main (int argc,char **argv) at whoami.c:26>:
    fprintf()
    who_am_i() <int who_am_i (void) at whoami.c:8>:
        getpwuid()
        geteuid()
        getenv()
        fprintf()
        printf()
```

This is a direct call graph showing *caller—callee* dependencies in the input file. Each line starts with a function name, followed by a pair of parentheses to indicate that it is a function. If this function is defined in one of the input files, the line continues by displaying, within a pair of angle brackets, a function *signature* and the location of its definition. If the function calls another functions, the line ends with a colon. For example, the line

```
main() <int main (int argc,char **argv) at whoami.c:25>:
```

shows that function `main` is defined in file `whoami.c` at line 25, as `int main (int argc, char **argv)`. Terminating colon indicates that `main` invokes other functions.

The lines following this one show which functions are called by `main`. Each such line is indented by a fixed amount of white space (by default, four spaces) for each nesting level.

Usually `cflow` prints a full function signature. However, sometimes you may wish to omit some part of it. Several options are provided for this purpose. To print signatures without function names, use `--omit-symbol-names` option. To omit argument list, use `--omit-arguments`. These options can be needed for a variety of reasons, one of them being to make the resulting graph more compact. To illustrate their effect, here is how would the first line of the above graph look if you had used both `--omit-` options:

```
main() <int () at whoami.c:25>:
```

By default, `cflow` starts outputting direct graph from the function called `main`. It is convenient when analyzing a set of input files comprising an entire `C` program. However, there are circumstances where a user would want to see only a part of the graph starting on particular function. One can instruct `cflow` to start output from the desiredfunction using `--main` (`-m`) command line option. Thus, running

```
cflow --main who_am_i whoami.c
```

on the above file will produce following graph:

```
who_am_i() <int who_am_i (void) at whoami.c:8>:
    getpwuid()
    geteuid()
    getenv()
    fprintf()
    printf()
```

Many programs (such as libraries or interpreters) define functions that are not directly reachable from the main function. To produce flow graph for all functions in the program, use the `--all` (`-A`) option. The output will then include separate flow graphs for each top-level function defined in the program. These graphs will be placed after the graph for `main` (if it exists), and will be ordered lexicographically by the function name.

When `--all` is used twice, graphs for all global functions (whether top-level or not) will be displayed.

To disable special handling of the `main` function, use the `--no-main` option.

3 Two Types of Flow Graphs.

In the previous chapter we have discussed *direct graphs*, displaying *caller—callee* dependencies. Another type of `cflow` output, called *reverse graph*, charts *callee—caller* dependencies. To produce a reverse graph, run `cflow` with `--reverse` (`-r`) command line option. For example, using a sample `whoami.c`:

```
$ cflow --reverse whoami.c
fprintf():
    who_am_i() <int who_am_i (void) at whoami.c:8>:
        main() <int main (int argc,char **argv) at whoami.c:26>
    main() <int main (int argc,char **argv) at whoami.c:26>
getenv():
    who_am_i() <int who_am_i (void) at whoami.c:8>:
        main() <int main (int argc,char **argv) at whoami.c:26>
geteuid():
    who_am_i() <int who_am_i (void) at whoami.c:8>:
        main() <int main (int argc,char **argv) at whoami.c:26>
getpwuid():
    who_am_i() <int who_am_i (void) at whoami.c:8>:
        main() <int main (int argc,char **argv) at whoami.c:26>
main() <int main (int argc,char **argv) at whoami.c:26>
printf():
    who_am_i() <int who_am_i (void) at whoami.c:8>:
        main() <int main (int argc,char **argv) at whoami.c:26>
who_am_i() <int who_am_i (void) at whoami.c:8>:
    main() <int main (int argc,char **argv) at whoami.c:26>
```

This output consists of several subgraphs, each describing callers for a particular function. Thus, the first subgraph tells that the function `fprintf` is called from two functions: `who_am_i` and `main`. First of them is, in turn, also called directly by `main`.

The first thing that draws attention in the above output is that the subgraph starting with `who_am_i` function is repeated several times. This is a *verbose* output. To make it brief, use `--brief` (`-b`) command line option. For example:

```
$ cflow --brief --reverse whoami.c
fprintf():
    who_am_i() <int who_am_i (void) at whoami.c:8>:
        main() <int main (int argc,char **argv) at whoami.c:26>
    main() <int main (int argc,char **argv) at whoami.c:26> [see 3]
getenv():
    who_am_i() <int who_am_i (void) at whoami.c:8>: [see 2]
geteuid():
    who_am_i() <int who_am_i (void) at whoami.c:8>: [see 2]
getpwuid():
    who_am_i() <int who_am_i (void) at whoami.c:8>: [see 2]
main() <int main (int argc,char **argv) at whoami.c:26> [see 3]
printf():
    who_am_i() <int who_am_i (void) at whoami.c:8>: [see 2]
who_am_i() <int who_am_i (void) at whoami.c:8>: [see 2]
```

In brief output, once a subgraph for a given function is written, subsequent instances of calls to that function contain only its definition and the *reference* to the output line where the expanded subgraph can be found.

If the output graph is large, it can be tedious to find out the required line number (unless you use *Emacs cflow-mode*, see Chapter 14 [Emacs], page 37). For such cases a special option `--number` (`-n`) is provided, which makes `cflow` begin each line of the output with a *reference number*, that is the ordinal number of this line in the output. With this option, the above output will look like:

```
$ cflow --number --brief --reverse whoami.c
    1 fprintf():
    2     who_am_i() <int who_am_i (void) at whoami.c:8>:
    3         main() <int main (int argc,char **argv) at whoami.c:26>
    4     main() <int main (int argc,char **argv) at whoami.c:26> [see 3]
    5 getenv():
    6     who_am_i() <int who_am_i (void) at whoami.c:8>: [see 2]
    7 geteuid():
    8     who_am_i() <int who_am_i (void) at whoami.c:8>: [see 2]
    9 getpwuid():
   10     who_am_i() <int who_am_i (void) at whoami.c:8>: [see 2]
   11 main() <int main (int argc,char **argv) at whoami.c:26> [see 3]
   12 printf():
   13     who_am_i() <int who_am_i (void) at whoami.c:8>: [see 2]
   14 who_am_i() <int who_am_i (void) at whoami.c:8>: [see 2]
```

Of course, `--brief` and `--number` options take effect for both direct and reverse flow graphs.

4 Various Output Formats.

The output format described in previous chapters is called *GNU Output*. Beside this, `cflow` is also able to produce output format defined in POSIX standard (The Open Group Base Specifications Issue 6: cflow utility (`http://www.opengroup.org/onlinepubs/009695399/utilities/cflow.html`)). In this format, each line of output begins with a *reference number*, i.e. the ordinal number of this line in the output, followed by indentation of fixed amount of columns per level (see [setting indentation], page 21). Following this are the name of the function, a colon and the function definition, if available. The function definition is followed by the location of the definition (file name and line number). Both definition and location are enclosed in angle brackets. If the function definition is not found, the line ends with an empty pair of angle brackets.

This output format is used when either a command line option `--format=posix` (`-f posix`) has been given, or environment variable `POSIXLY_CORRECT` was set.

The output graph in POSIX format for our sample `whoami.c` file will look as follows:

```
$ cflow --format=posix whoami.c
    1 main: int (int argc,char **argv), <whoami.c 26>
    2     fprintf: <>
    3     who_am_i: int (void), <whoami.c 8>
    4         getpwuid: <>
    5         geteuid: <>
    6         getenv: <>
    7         fprintf: <>
    8         printf: <>
```

It is not clear from the POSIX specification whether the output should contain argument lists in function declarations, or not. By default `cflow` will print them. However, some programs, analyzing `cflow` output expect them to be absent. If you use such a program, add `--omit-arguments` option to `cflow` command line (see [omit signature parts], page 4).

Future versions of `cflow` will offer more output formats, including XML and HTML outputs. Currently, you can use `VCG` tool (`http://rw4.cs.uni-sb.de/users/sander/html/gsvcg1.html`) to create graphical representation of the produced graphs. To transform `cflow` output to `xvcg` input syntax, use `cflow2vcg` program (`http://cflow2vcg.sourceforge.net/`). Both programs are available under GPL.

`Cflow2vcg` expects POSIX call graphs, indented with exactly one horizontal tabulation character per nesting level, with an additional tab character for zeroth level and without argument lists in function declaration. So, to produce an output suitable for `cflow2vcg`, invoke `cflow` as follows[1]:

```
cflow --format=posix --omit-arguments \
      --level-indent='0=\t' --level-indent='1=\t' \
      --level-indent=start='\t'
```

You can use the following script to visualize call graphs using the three tools:

[1] (See Chapter 8 [ASCII Tree], page 21, for the detailed description of `--level-indent` option

```
#! /bin/sh

cflow --format=posix --omit-arguments \
      --level-indent='0=\t' --level-indent='1=\t' \
      --level-indent=start='\t' $* |
  cflow2vcg | xvcg -
```

5 Handling Recursive Calls.

Sometimes programs contain functions that recursively call themselves. GNU output format provides a special indication for such functions. The definition of the recursive function is marked with an '`(R)`' at the end of line (before terminating colon). Subsequent recursive calls to this function are marked with a '`(recursive: see` *`refline`*`)`' at the end of line. Here, *refline* stands for the reference line number where the *recursion root* definition was displayed.

To illustrate this, let's consider the following program, that prints recursive listing of a directory, allowing to cut off at the arbitrary nesting level:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdlib.h>
#include <string.h>

/* Return true if file NAME is a directory. */
static int
isdir (char *name)
{
  struct stat st;

  if (stat (name, &st))
    {
      perror (name);
      return 0;
    }
  return S_ISDIR (st.st_mode);
}

static char *ignored_names[] = { ".", "..", NULL };

/* Return true if NAME should not be recursed into */
int
ignorent (char *name)
{
  char **p;
  for (p = ignored_names; *p; p++)
    if (strcmp (name, *p) == 0)
      return 1;
  return 0;
}
```

```
int max_level = -1;

/* Print contents of the directory PREFIX/NAME.
   Prefix each output line with LEVEL spaces. */
void
printdir (int level, char *name)
{
  DIR *dir;
  struct dirent *ent;
  char cwd[512];

  if (!getcwd(cwd, sizeof cwd))
    {
      perror ("cannot save cwd\n");
      _exit (1);
    }
  chdir (name);
  dir = opendir (".");
  if (!dir)
    {
      perror (name);
      _exit (1);
    }

  while ((ent = readdir (dir)))
    {
      printf ("%*.*s%s", level, level, "", ent->d_name);
      if (ignorent (ent->d_name))
        printf ("\n");
      else if (isdir (ent->d_name))
        {
          printf ("/");
          if (level + 1 == max_level)
            putchar ('\n');
          else
            {
              printf (" contains:\n");
              printdir (level + 1, ent->d_name);
            }
        }
      else
        printf ("\n");
    }
  closedir (dir);
  chdir (cwd);
}
```

```
int
main (int argc, char **argv)
{
  if (argc < 2)
    {
      fprintf (stderr, "usage: d [-MAX] DIR [DIR...]\n");
      return 1;
    }

  if (argv[1][0] == '-')
    {
      if (!(argv[1][1] == '-' && argv[1][2] == 0))
        max_level = atoi (&argv[1][1]);
      --argc;
      ++argv;
    }

  while (--argc)
    printdir (0, *++argv);

  return 1;
}
```

Running `cflow` on this program produces the following graph:

```
$ cflow --number d.c
    1 main() <int main (int argc,char **argv) at d.c:85>:
    2     fprintf()
    3     atoi()
    4     printdir() <void printdir (int level,char *name) at d.c:42> (R):
    5         getcwd()
    6         perror()
    7         chdir()
    8         opendir()
    9         readdir()
   10         printf()
   11         ignorent() <int ignorent (char *name) at d.c:28>:
   12             strcmp()
   13         isdir() <int isdir (char *name) at d.c:12>:
   14             stat()
   15             perror()
   16             S_ISDIR()
   17         putchar()
   18         printdir()
                <void printdir (int level,char *name) at d.c:42>
                (recursive: see 4)
   19         closedir()
```

The `printdir` description in line 4 shows that the function is recursive. The recursion call is shown in line 18.

6 Controlling Symbol Types

An alert reader has already noticed something strange in the above output: the function `_exit` is missing, although according to the source file it is called twice by `printdir`. It is because by default `cflow` omits from its output all symbols beginning with underscore character. To include these symbols as well, specify `-i _` (or `--include _`) command line option. Continuing our example:

```
$ cflow --number -i _ d.c
    1 main() <int main (int argc,char **argv) at d.c:85>:
    2     fprintf()
    3     atoi()
    4     printdir() <void printdir (int level,char *name) at d.c:42> (R):
    5         getcwd()
    6         perror()
    7         _exit()
    8         chdir()
    9         opendir()
   10         readdir()
   11         printf()
   12         ignorent() <int ignorent (char *name) at d.c:28>:
   13             strcmp()
   14         isdir() <int isdir (char *name) at d.c:12>:
   15             stat()
   16             perror()
   17             S_ISDIR()
   18         putchar()
   19         printdir()
                <void printdir (int level,char *name) at d.c:42>
                (recursive: see 4)
   20         closedir()
```

In general, `--include` takes an argument specifying a list of *symbol classes*. Default option behavior is to include the requested classes to the output. If the argument begins with a minus or caret sign, this behavior is reversed and the requested symbol classes are excluded from the output.

The symbol class '`_`' includes symbols whose names begin with an underscore. Another useful symbol class is '`s`', representing *static functions or data*. By default, static functions are always included in the output. To omit them, one can give `-i ^s` (or `-i -s`[1]) command line option. Our sample program `d.c` defines static function `isdir`, running `cflow -i ^s`, completely omits this function and its callees from the resulting graph:

```
$ cflow --number -i ^s d.c
    1 main() <int main (int argc,char **argv) at d.c:85>:
    2     fprintf()
    3     atoi()
```

[1] Notice that `-i -s` is a single option, in spite of `-s` beginning with a minus sign. Since this might be confusing, we prefer using '`^`' instead of '`-`' to denote symbol exclusion.

```
    4     printdir() <void printdir (int level,char *name) at d.c:42> (R):
    5         getcwd()
    6         perror()
    7         chdir()
    8         opendir()
    9         readdir()
   10         printf()
   11         ignorent() <int ignorent (char *name) at d.c:28>:
   12             strcmp()
   13         putchar()
   14         printdir()
                <void printdir (int level,char *name) at d.c:42>
                (recursive: see 4)
   15         closedir()
```

Actually, the exclusion sign (‘^’ or ‘-’) can be used any place in `-i` argument, not only at the beginning. Thus, option `-i _^s` means “*include symbols, beginning with underscore and exclude static functions*”. Several `-i` options accumulate, so the previous example can also be written as `-i _ -i ^s`.

It is important to notice that by default `cflow` graphs contain only functions. You can, however, request displaying variables as well, by using symbol class ‘x’. This class contains all *data symbols*, both global and static, so to include these in the output, use option `-i x`. For example:

```
$ cflow --number -i x d.c
    1 main() <int main (int argc,char **argv) at d.c:85>:
    2     fprintf()
    3     stderr
    4     max_level <int max_level at d.c:37>
    5     atoi()
    6     printdir() <void printdir (int level,char *name) at d.c:42> (R):
    7         DIR
    8         dir
    9         getcwd()
   10         perror()
   11         chdir()
   12         opendir()
   13         readdir()
   14         printf()
   15         ignorent() <int ignorent (char *name) at d.c:28>:
   16             ignored_names <char *ignored_names[] at d.c:24>
   17             strcmp()
   18         isdir() <int isdir (char *name) at d.c:12>:
   19             stat()
   20             perror()
   21             S_ISDIR()
   22             NULL
   23         max_level <int max_level at d.c:37>
```

```
24          putchar()
25          printdir()
              <void printdir (int level,char *name) at d.c:42>
              (recursive: see 6)
26          closedir()
```

Now, lines 3, 4, 16 and 23 show data symbols, with their definitions when available. Notice, however, lines 7 and 8. Why both type name `DIR` and automatic variable `dir` are listed as data?

To answer this question, let's first describe the `cflow` notion of symbols. The program keeps its *symbol tables*, which are initially filled with `C` predefined keywords. When parsing input files, `cflow` updates these tables. In particular, upon encountering a `typedef`, it registers the defined symbol as a *type*.

Now, `DIR` is not declared in `d.c`, so `cflow` has no way of knowing it is a data type. So, it supposes it is a variable. But then the input:

```
DIR *dir;
```

is parsed as an *expression*, meaning "multiply `DIR` by `dir`".

Of course, it is wrong. There are two ways to help `cflow` out of this confusion. You can either explicitly declare `DIR` as data type, or let `cflow` run preprocessor, so it sees the contents of the include files and determines it by itself. Running preprocessor is covered by the next chapter (see Chapter 7 [Preprocessing], page 19). In the present chapter we will concentrate on the first method.

The command line option `--symbol` (`-s`) declares a *syntactic class* of the symbol. Its argument consists of two strings separated by a colon:

```
--symbol sym:class
```

The first string, *sym* is a `C` identifier to be recorded in the symbol table. The second string, *class*, specifies a class to be associated with this symbol. In particular, if *class* is '`type`', the symbol *sym* will be recorded as a `C` type definition. Thus, to fix the above output, run:

```
$ cflow --number -i x --symbol DIR:type d.c
```

Another important symbol type is a *parameter wrapper*. It is a kind of a macro, often used in sources that are meant to be compatible with pre-ANSI compilers to protect parameter declarations in function prototypes. For example, in the declaration below, taken from `/usr/include/resolv.h`, `__P` is a parameter wrapper:

```
void res_npquery __P((const res_state, const u_char *, int, FILE *));
```

For `cflow` to be able to process such declarations, declare `__P` as a wrapper, for example:

```
cflow --symbol __P:wrapper *.c
```

In both examples above the reason for using the `--symbol` option was that `cflow` was unable to determine what the given symbol was, either because it did not see the type definition, as it was in case with '`DIR`', or because the macro definition was not expanded. Both cases are better solved by using *preprocess mode*, described in the next chapter. Nevertheless, even with preprocessor, the `--symbol` option remains useful, as shown in the following sections.

6.1 Syntactic classes

Generally speaking, the *syntactic class* of a symbol defines where in the C code this symbol can legitimately appear. There are following classes:

keyword
kw
: A keyword, like '`if`', '`when`' etc.

modifier
: Type modifier, i.e. the symbol appearing after a data type to modify its meaning, like '`*`'.

qualifier
: Declaration qualifier. Can appear both before C declaration (much like '`static`' or '`extern`') and after a data type (like modifiers).

  You would usually declare a `gcc` keyword '`__extension__`' as a qualifier:

  ```
  --symbol __extension__:qualifier
  ```

identifier
: A C identifier.

type
: A C data type, like '`int`', '`char`', etc.

wrapper
: That has two meanings. First, it can be used to declare parameter wrappers when running `cflow` without preprocessor. This usage was described above. Second, it indicates any symbol that can appear in a declaration either before an identifier or before a terminating semicolon and optionally followed by a parenthesized expression list.

  We recommended to use this class for the `gcc` '`__attribute__`' keyword.

6.2 Symbol aliases

Yet another use for the `--symbol` option is to define *symbol aliases*. An alias is a token that behaves exactly as the symbol it refers to. Alias is declared using the following construct:

```
--symbol newsym:=oldsym
```

As a result of this option, the symbol *newsym* is declared to be the equivalent of *oldsym*.

Symbol aliasing can be regarded as defining the symbol class by example of another symbol. It is useful for some special keywords, such as '`__restrict`':

```
--symbol __restrict:=restrict
```

6.3 GCC Initialization

The following is a recommended set of `cflow` initialization options for use with `gcc`. We suggest you to put them in your `cflow.rc` file (see Chapter 10 [Configuration], page 27).

```
--symbol __inline:=inline
--symbol __inline__:=inline
--symbol __const__:=const
--symbol __const:=const
--symbol __restrict:=restrict
--symbol __extension__:qualifier
--symbol __attribute__:wrapper
--symbol __asm__:wrapper
--symbol __nonnull:wrapper
--symbol __wur:wrapper
```

7 Running Preprocessor

`Cflow` can preprocess input files before analyzing them, the same way `cc` does before compiling. Doing so allows `cflow` to correctly process all symbol declarations, thus avoiding the necessity to define special symbols using `--symbol` option, described in the previous chapter. To enable preprocessing, run the utility with `--cpp` (`--preprocess`) command line option. For our sample file `d.c`, this mode gives:

```
$ cflow --cpp -n d.c
    1 main() <int main (int argc,char **argv) at d.c:85>:
    2     fprintf()
    3     atoi()
    4     printdir() <void printdir (int level,char *name) at d.c:42> (R):
    5         getcwd()
    6         perror()
    7         chdir()
    8         opendir()
    9         readdir()
   10         printf()
   11         ignorent() <int ignorent (char *name) at d.c:28>:
   12             strcmp()
   13         isdir() <int isdir (char *name) at d.c:12>:
   14             stat()
   15             perror()
   16         putchar()
   17         printdir()
                <void printdir (int level,char *name) at d.c:42>
                (recursive: see 4)
   18         closedir()
```

Compare this graph with the one obtained without `--cpp` option (see [sample flowchart], page 13). As you see, the reference to `S_ISDIR` is gone: the macro has been expanded. Now, try running `cflow --cpp --number -i x d.c` and compare the result with the graph obtained without preprocessing (see [x flowchart], page 16). You will see that it produces correct results without using `--symbol` option.

By default `--cpp` runs `/usr/bin/cpp`. If you wish to run another preprocessor command, specify it as an argument to the option, after an equal sign. For example, `cflow --cpp='cc -E'` will run the `C` compiler as a preprocessor.

8 Using ASCII Art to Produce Flow Graphs.

You can configure the exact appearance of `cflow` output flow graph using `--level-indent` option. The simplest use for this option is to change the default indentation per nesting level. To do so, give the option a numeric argument specifying the number of columns to indent for each nesting level. For example, the following command sets the indentation level to 2, which is half of the default:

```
cflow --level-indent 2 d.c
```

It can be used, for instance, to keep the graph within the page margins.

However, `--level-indent` can do much more than that. Each line in the flow graph consists of the following graphical elements: a *start marker*, an *end marker*, with several *indent fills* between them. By default, both start and end markers are empty, and each indent fill contains four spaces.

If the argument to `--level-indent` option has the form *element*=*string*, it specifies a character string that should be output in place of a given graph element. The element names are:

start	Start marker
0	Indent fill 0
1	Indent fill 1
end0	End marker 0
end1	End marker 1

Why are there two kinds of indent fills and end markers? Remember that the flow graph represents a call tree, so it contains terminal nodes (*leaves*), i.e. the calls that end a function, and non-terminal nodes (the calls followed by another ones on the same nesting level). The *end marker 0* is for non-terminal nodes, and *end marker 1* is for terminal nodes.

As for indent fills, *indent fill 1* is used to represent graph edge, whereas *fill 0* is used to keep the output properly aligned.

To demonstrate this, let's consider following sample program:

```
/* foo.c */
int
main()
{
    f();
    g();
    f();
}

int
f()
{
    i = h();
}
```

Now, let's represent line elements by the following strings:

start	'`::`'

0	' ' (two spaces)
1	'| ' (a vertical bar and a space)
end0	'+-'
end1	'\-'

The corresponding command line will be: `cflow --level begin=:: --level '0= ' --level '1=| ' --level end0='+-' --level end1='\\-' foo.c`. Notice escaping the backslash characters in `end1`: generally speaking, *string* in `--level-option` can contain usual `C` escape sequences, so the backslash character itself must be escaped. Another shortcut, allowed in *string* is the notation `CxN`, where *C* is any single character and *N* is a decimal number. This notation means "*repeat character C N times*". However, character 'x' looses its special meaning if used at the beginning of the string.

This command will produce the following output:

```
::+-main() <int main () at foo.c:3>:
::  +-f() <int f () at foo.c:11>:
::  | \-h()
::  \-g()
```

Thus, we obtained an *ASCII art* representation of the call tree. GNU `cflow` provides a special option `--tree` (`-T`), which is a shortcut for `--level '0= ' --level '1=| ' --level end0='+-' --level end1='\\-'`. The following is an example of flow graph produced with this option. The source file `wc.c` is a simple implementation of UNIX `wc` command, See Appendix A [Source of wc command], page 41.

```
$ cflow --tree --brief --cpp wc.c
+-main() <int main (int argc,char **argv) at wc.c:127>
  +-errf() <void errf (char *fmt,...) at wc.c:34>
  | \-error_print()
  |     <void error_print (int perr,char *fmt,va_list ap) at wc.c:22>
  |   +-vfprintf()
  |   +-perror()
  |   +-fprintf()
  |   \-exit()
  +-counter() <void counter (char *file) at wc.c:108>
  | +-fopen()
  | +-perrf() <void perrf (char *fmt,...) at wc.c:46>
  | | \-error_print()
  | |    <void error_print (int perr,char *fmt,va_list ap)
  | |    at wc.c:22> [see 3]
  | +-getword() <int getword (FILE *fp) at wc.c:78>
  | | +-feof()
  | | \-isword() <int isword (unsigned char c) at wc.c:64>
  | |   \-isalpha()
  | +-fclose()
  | \-report()
  |     <void report (char *file,count_t ccount,
  |       count_t wcount,count_t lcount) at wc.c:57>
  |   \-printf()
  \-report()
       <void report (char *file,count_t ccount,
         count_t wcount,count_t lcount) at wc.c:57> [see 17]
```

9 Cross-Reference Output.

GNU `cflow` is also able to produce *cross-reference listings*. This mode is enabled by `--xref` (`-x`) command line option. Cross-reference output lists each symbol occurrence on a separate line. Each line shows the identifier and the source location where it appears. If this location is where the symbol is defined, it is additionally marked with an asterisk and followed by the definition. For example, here is a fragment of a cross-reference output for `d.c` program:

```
printdir * d.c:42 void printdir (int level,char *name)
printdir   d.c:74
printdir   d.c:102
```

It shows that the function `printdir` is defined in line 42 and referenced twice, in lines 74 and 102.

The symbols included in cross-reference listings are controlled by `--include` option (see [–include], page 15). In addition to character classes discussed in chapter “Controlling Symbol Types” (see Chapter 6 [Symbols], page 15), an additional symbol class `t` controls listing of type names defined by `typedef` keyword.

10 Configuration Files and Variables.

As shown in the previous chapters, GNU `cflow` is highly configurable. Different command line options have different effects, as specifying new operation modes or altering some aspects of the output. You will likely use some options frequently, while you will use others from time to time, or not at all (See Chapter 12 [Options], page 31, for a full list of options).

The `CFLOW_OPTIONS` environment variable specifies default options to be placed in front of any explicit options. For example, if you set `CFLOW_OPTIONS="--format=posix --cpp"` in your `.profile`, `cflow` will behave as if the two options `--format=posix` and `--cpp` had been specified before any explicit options.

There is also another possibility to specify your default options. After incorporating eventual content of `CFLOW_OPTIONS` variable, `cflow` checks the value of the environment variable `CFLOWRC`. This value, if not empty, specifies the name of the *configuration file* to read. If `CFLOWRC` is not defined or is empty, the program attempts to read file `.cflowrc` in the user's home directory. It is not an error if any of these files does not exist. However, if the file does exist but cannot be processed, `cflow` will issue an explicit error message.

The configuration file is read line by line. Empty lines and lines beginning with usual `shell` comment character ('`#`') are ignored. Otherwise, the line is split into *words*, the same way `shell` does, and the resulting words are placed in the command line after any options taken from `CFLOW_OPTIONS` variable, but before any explicit options.

Pay attention when using such options as `-D` in the configuration file. The value of the `-D` option will be added to the preprocessor command line and will be processed by the shell, so be careful to properly quote its argument. The rule of thumb is: "*use the same quoting you would have used in the shell command line*". For example, to run `cc -E` as a preprocessor, you can use the following configuration file:

```
--cpp='cc -E'
-DHAVE_CONFIG_H
-D__extension__\\\(c\\\)=
```

By the way, the above example shows a way of coping with the '`__extension__()`' construct used by `gcc`, i.e. by defining it to an empty string.

It may sometimes be necessary to cancel the effect of a command line option. For example, you might specify `--brief` in your configuration file, but then occasionally need to obtain verbose graph. To cancel the effect of any GNU `cflow` option that does not take arguments, prepend '`no-`' to the corresponding long option name. Thus, specifying `--no-brief` cancels the effect of the previous `--brief` option.

11 Using cflow in Makefiles.

If you wish to use `cflow` to analyze your project sources, `Makefile` or `Makefile.am` is the right place to do so. In this chapter we will describe a generic rule for `Makefile.am`. If you do not use `automake`, you can deduce the rule for plain `Makefile` from this one.

Here is a check list of steps to do to set up a `Makefile.am` framework:

– If you use a configuration file, add it to `EXTRA_DIST` variable.
– Add variable `CFLOW_FLAGS` with any special `cflow` options you wish to use. The variable can be empty, its main purpose is making it possible to override `cflow` options by running `make CFLOW_FLAGS=... chart`.
– For each *program* from your *dir*_PROGRAMS list, for which you want to generate a flow chart, add the following statements:

```
program_CFLOW_INPUT=$(program_OBJECTS:.$(OBJEXT)=.c)
program.cflow: program_CFLOW_INPUT cflow.rc Makefile
CFLOWRC=path-to-your-cflow.rc \
 cflow -oprogram.cflow $(CFLOW_FLAGS) $(DEFS) \
                   $(DEFAULT_INCLUDES) $(INCLUDES) $(AM_CPPFLAGS) \
           $(CPPFLAGS) \
        $(program_CFLOW_INPUT)
```

Replace *program* with program name and *path-to-your-cflow.rc* with the full file name of your `cflow.rc` file (if any). If you do not wish to use preprocessing, remove from the `cflow` command line all variables, except `CFLOW_FLAGS`.

– If there are several programs built by this `Makefile.am`, you may wish to add a special rule, allowing to create all flow charts with a single command, for example:

```
flowcharts: prog1.cflow prog2.cflow ...
```

As an example, here are the relevant statements which we use in `cflow` `src/Makefile.am`:

```
EXTRA_DIST=cflow.rc

CFLOW_FLAGS=-i^s
cflow_CFLOW_INPUT=$(cflow_OBJECTS:.$(OBJEXT)=.c)
cflow.cflow: $(cflow_CFLOW_INPUT) cflow.rc Makefile
CFLOWRC=$(top_srcdir)/src/cflow.rc \
 cflow -ocflow.cflow $(CFLOW_FLAGS) $(DEFS) \
                   $(DEFAULT_INCLUDES) $(INCLUDES) $(AM_CPPFLAGS) \
           $(CPPFLAGS) \
        $(cflow_CFLOW_INPUT)
```

12 Complete Listing of `cflow` Options.

This chapter contains an alphabetical listing of all `cflow` command line options, with brief descriptions and cross references to more in-depth explanations in the body of the manual. Both short and long option forms are listed, so you can use this table as a quick reference.

Most of the options have a *negation counterpart*, an option with a reverse meaning. The name of a negation option is formed by prefixing the corresponding long option name with a `no-`. This feature is provided to cancel default options specified in the configuration file.

In the table below, options with negation counterparts are marked with a bullet (•).

`-A`
`--all`
: Produce graphs for all global functions in the program. Use this option if your program contains functions, which are not directly reachable from `main` (see [start symbol], page 4).

`-a`
`--ansi`
: • Assume input to be written in ANSI `C`. Currently this means disabling code that parses *K&R function declarations*. This might speed up the processing in some cases.

`-b`
`--brief`
: • Brief output. See [–brief], page 7.

`--cpp[=`*command*`]`
: • Run the specified preprocessor command. See Chapter 7 [Preprocessing], page 19.

`-D` *name*`[=`*defn*`]`
`--define=`*name*`[=`*defn*`]`
: Predefine *name* as a macro. Implies `--cpp` (see Chapter 7 [Preprocessing], page 19).

`-d` *number*
`--depth=`*number*
: Set the depth at which the flow graph is cut off. For example, `--depth=5` means the graph will contain function calls up to the 5th nesting level.

`--debug[=`*number*`]`
: Set debugging level. The default *number* is 1. Use this option if you are developing and/or debugging `cflow`.

`--emacs`
: • Prepend the output with a line telling Emacs to use `cflow` mode when visiting this file. Implies `--format=gnu`. See [–emacs], page 37.

`-f` *name*
`--format=`*name*
: Use given output format *name*. Valid names are `gnu` (see [GNU Output Format], page 4) and `posix` (see [POSIX Output Format], page 9).

`-?`
`--help`
: Display usage summary with short explanation for each option.

`-I dir`
`--include-dir=dir`
: Add the directory *dir* to the list of directories to be searched for header files. Implies `--cpp` (see Chapter 7 [Preprocessing], page 19).

`-i spec`
`--include=spec`
: Control the number of included symbols. *Spec* is a string consisting of characters, specifying what class of symbols to include in the output. Valid *spec* symbols are:

  `-`
  `^` Exclude symbols denoted by the following letters.

  `+` Include symbols denoted by the following letters (default).

  `_` Symbols whose names begin with an underscore.

  s Static symbols.

  t Typedefs (for cross-references only, see Chapter 9 [Cross-References], page 25).

  x All data symbols, both external and static.

  For more information, See Chapter 6 [Symbols], page 15.

`-l` See [–print-level], page 33.

`--level-indent=string`
: Use *string* when indenting to each new level. See Chapter 8 [ASCII Tree], page 21.

`-m name`
`--main=name`
: Assume main function to be called *name*. See [start symbol], page 4.

`--no-main`
: Assume there's no main function in the program. This option has the same effect as `--all`, except that, if the program defines the `main` function, it will be treated as any other functions.

`-n`
`--number` • Print line numbers. See [–number], page 8.

`-o file`
`--output=file`
: Set output file name. Default is '`-`', meaning standard output.

`--omit-arguments`
: • Do not print argument lists in function declarations. See [omit signature parts], page 4.

`--omit-symbol-names`
: • Do not print symbol names in declarations. See [omit signature parts], page 4. This option is turned on in '`posix`' output mode (see [POSIX Output Format], page 9.

`-r`
`--reverse`
: • Print reverse call graph. See Chapter 3 [Direct and Reverse], page 7.

`-x`
`--xref`
: • Produce cross-reference listing only. See Chapter 9 [Cross-References], page 25.

`-p` *number*
`--pushdown=`*number*
: Set initial token stack size to *number* tokens. Default is 64. The token stack grows automatically when it needs to accommodate more tokens than its current size, so it is seldom necessary to use this option.

`--preprocess[=`*command*`]`
: Run the specified preprocessor command. See [–cpp], page 31.

`-s` *sym*`:`*class*
`--symbol=`*sym*`:`*class*
`--symbol=`*newsym*`:=`*oldsym*
: In the first form, registers symbol *sym* in the syntactic class *class*. Valid class names are: '`keyword`' (or '`kw`'), '`modifier`', '`qualifier`', '`identifier`', '`type`', '`wrapper`'. Any unambiguous abbreviation of the above is also accepted. See Section 6.1 [Syntactic classes], page 18.

In the second form (with the '`:=`' separator), defines *newsym* as an alias to *oldsym*. See Section 6.2 [Symbol aliases], page 18.

See Section 6.3 [GCC Initialization], page 18, for a practical example of using this option.

`-S`
`--use-indentation`
: • Use source file indentation as a hint. Currently this means that the closing curly brace ('`}`') in the column zero forces `cflow` to close current function definition. Use this option sparingly, it may cause misinterpretation of some sources.

`-U` *name*
`--undefine=`*name*
: Cancel any previous definition of *name*. Implies `--cpp` (see Chapter 7 [Preprocessing], page 19).

`--print-level`
`-l`
: • Print nesting level along with the call graph. The level is printed after output line number (if `--number` or `--format=posix` is used, enclosed in curly braces.

`-T`
`--tree`
: • Use ASCII art to print graph. See Chapter 8 [ASCII Tree], page 21.

`--usage`
: Give a short usage message.

`-v`
`--verbose`

• Verbosely list any errors encountered in the input files. The `cflow` notion of an error does not match that of `C` compiler, so by default error messages are turned off. It is useful to enable them if you suspect that `cflow` misinterprets the sources.

`-V`
`--version`

Print program version.

13 Exit Codes

0	Successful completion.
1	Fatal error occurred.
2	Some input files cannot be read or parsed.
3	Command line usage error.

14 Using `cflow` with GNU Emacs.

GNU `cflow` comes with an `emacs` module providing a major mode for visiting flow charts in GNU Emacs. If you have a working `emacs` on your machine, the module will be installed somewhere in your Emacs `load-path`. To load the module at startup, add the following lines to your `.emacs` or `site-start.el` file:

```
(autoload 'cflow-mode "cflow-mode")
(setq auto-mode-alist (append auto-mode-alist
                              '(("\\.cflow$" . cflow-mode))))
```

The second statement associates `cflow-mode` with any file having suffix `.cflow`. If you prefer to have another suffix for flow graph files, use it instead. You can also omit this option, if you do not use any special suffix for your graph files. In this case we recommend using `--emacs` command line option. This option generates the first line telling Emacs to use `cflow` major mode when visiting the file.

The buffer opened in `cflow` mode is made read-only. The following key bindings are defined:

`E` Temporarily exits from `cflow` mode and allows you to edit the graph file. To resume `cflow` mode type `M-x` cflow-mode `RET`. This option is provided mainly for debugging purposes. We do not recommend you to edit chart files, since this will change line numbering and thus prevent `cflow` mode from correctly tracing line references.

`x` Go to expansion of the current graph vertex. Use this key if the point stands on a line ending with '`[see N]`' reference. It will bring you directly to the referenced line. Use `exchange-point-and-mark` (by default `C-x C-x`) to return to the line you examined.

`R` If the point is standing on a recursive function, go to the next recursion. Sets mark.

`r` If the point is standing on a recursive function, return to its definition (a *recursion root*). Sets mark.

`s` Visit the referenced source file and find the function definition.

15 How to Report a Bug

Send bug reports via electronic mail to `bug-cflow@gnu.org`.

As the purpose of bug reporting is to improve software, please be sure to include maximum information when reporting a bug. The minimal information needed is:

- Version of the package you are using.
- Compilation options used when configuring the package.
- Detailed description of the bug.
- Conditions under which the bug appears (command line options, input file contents, etc.)

Appendix A Source of the wc command

The source file wc.c, used to produce sample ASCII tree graph (see [ascii tree], page 22).

```
/* Sample implementation of wc utility. */

#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>

typedef unsigned long count_t;  /* Counter type */

/* Current file counters: chars, words, lines */
count_t ccount;
count_t wcount;
count_t lcount;

/* Totals counters: chars, words, lines */
count_t total_ccount = 0;
count_t total_wcount = 0;
count_t total_lcount = 0;

/* Print error message and exit with error status. If PERR is not 0,
   display current errno status. */
static void
error_print (int perr, char *fmt, va_list ap)
{
  vfprintf (stderr, fmt, ap);
  if (perr)
    perror (" ");
  else
    fprintf (stderr, "\n");
  exit (1);
}

/* Print error message and exit with error status. */
static void
errf (char *fmt, ...)
{
  va_list ap;

  va_start (ap, fmt);
  error_print (0, fmt, ap);
  va_end (ap);
}

/* Print error message followed by errno status and exit
   with error code. */
```

```
static void
perrf (char *fmt, ...)
{
  va_list ap;

  va_start (ap, fmt);
  error_print (1, fmt, ap);
  va_end (ap);
}

/* Output counters for given file */
void
report (char *file, count_t ccount, count_t wcount, count_t lcount)
{
  printf ("%6lu %6lu %6lu %s\n", lcount, wcount, ccount, file);
}

/* Return true if C is a valid word constituent */
static int
isword (unsigned char c)
{
  return isalpha (c);
}

/* Increase character and, if necessary, line counters */
#define COUNT(c)       \
      ccount++;        \
      if ((c) == '\n') \
        lcount++;

/* Get next word from the input stream. Return 0 on end
   of file or error condition. Return 1 otherwise. */
int
getword (FILE *fp)
{
  int c;
  int word = 0;

  if (feof (fp))
    return 0;

  while ((c = getc (fp)) != EOF)
    {
      if (isword (c))
        {
          wcount++;
          break;
```

```
        }
      COUNT (c);
    }

  for (; c != EOF; c = getc (fp))
    {
      COUNT (c);
      if (!isword (c))
        break;
    }

  return c != EOF;
}

/* Process file FILE. */
void
counter (char *file)
{
  FILE *fp = fopen (file, "r");

  if (!fp)
    perrf ("cannot open file ‘%s’", file);

  ccount = wcount = lcount = 0;
  while (getword (fp))
    ;
  fclose (fp);

  report (file, ccount, wcount, lcount);
  total_ccount += ccount;
  total_wcount += wcount;
  total_lcount += lcount;
}

int
main (int argc, char **argv)
{
  int i;

  if (argc < 2)
    errf ("usage: wc FILE [FILE...]");

  for (i = 1; i < argc; i++)
    counter (argv[i]);

  if (argc > 2)
    report ("total", total_ccount, total_wcount, total_lcount);
```

```
  return 0;
}
```

Appendix B GNU Free Documentation License

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

   A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

   The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

B.1 ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Concept Index

This is a general index of all issues discussed in this manual

_

0

1

A

B

C

D

E

F

G

I

M

O

P

R

S

X