The `optparse.c` module

R.A. Litherland

1 Introduction

This module, consisting of the files `optparse.c` and `optparse.h`, provides functions to parse the optional arguments to a program, and to print a help message listing all options. It was inspired the Python module optparse, which is referred to below as `optparse.py`, and attempts to make it easy to get most of the functionality provided by `optparse.py` in a C program. Options can have one or more short forms such as `-h`, a long form such as `--help`, or both, but only one long form is allowed. Short forms can be combined in a single argument, and long forms can be abbreviated if the result is unambiguous (and at least two characters long). An option may require futher information (e.g., a filename), which I call the *value* of the option. (Options requiring more than one value can be implemented using the function `opt_remainder`; outside the description of that function, this possibility is ignored.) When an option given in short form needs a value, the value is the rest of the option argument if that has at least one character, and the next argument if not. The value of an option given in long form can be given in the same argument, as in `--file=foo.txt`, or as the next argument.

Terminology.

The words "argument", "help" and "usage" will, I'm afraid, be somewhat overused. An *argument* is either what you pass to a C function, or one of a list of strings you pass to the `opt_parse` function. The list is typically the `argv` of your `main` function, so I will sometimes refer to these strings as *program arguments.* However, as mentioned above, I refer to the 'value' rather than the 'argument' of an option. Program arguments are divided into three classes: *option arguments*, which specify one or more options; *value arguments*, which supply the values of options, and *positional arguments*, which are all the rest.

The *program help message* is everything that gets printed when the user gives the help option to your program. Figure 1 shows an example, taken from the documentation of `optparse.py` (see pp. 279–280 of the Python Library Reference). The first line,

```
usage: fudd [options] arg1 arg2
```

is the *program usage string.* An item like

```
  -h, --help           print this help message and exit
```

is an *option usage string*, which is made up of the *option names* (`-h, --help`) and the *help text.* Everything else is *filler.*

Finally, by an *empty string* I mean a value `s` of type `char*` satisfying `!s || !*s`.

```
leibniz:~/optparse> ./fudd -h
usage: fudd [options] arg1 arg2
options:
  -h, --help          print this help message and exit
  -v, --verbose       make lots of noise [default]
  -q, --quiet         be vewwy quiet (I'm hunting wabbits)
  -fFILE, --file=FILE  write output to FILE
  -mMODE, --mode=MODE  interaction mode: one of 'novice', 'intermediate'
                      [default], 'expert'
  Dangerous Options:
    Caution: use of these options is at your own risk.  It is believed that
    some of them bite.
    -g                Group option.
```

Figure 1: A program help message.

In the following sections, I describe all functions, types and macros declared in `optparse.h`; their names all begin with `opt_` or `OPT_`.

2 Specifying options

The type struct opt_spec.
Synopsis. `struct opt_spec {`

```
        int (*action)(char *, void *);
        const char *sf, *lf, *metavar, *help;
        void *data;
    };
```

Description. This struct is used to specify options to the `opt_parse` function. Suppose that `o` has type `struct opt_spec`. A null pointer in `o.action` marks the end of an array, and the special value `opt_text` signals that this item does not specify an option, but just provides text to be included in the program help message; this case is described later. Otherwise, `o.action` is called when the option specified is encountered, with its second argument set to `o.data`. For an action you define, `o.data` can be anything you like[1]. Several pre-defined functions that can be used as the `o.action` field are described in §5, and what they expect as their second argument is specified there. For an option not taking a value, the first argument will be `NULL`; for one taking a value, it will point to the value. Note that storing this pointer for later use is problematical; see the description of `opt_store_str` in §5. A non-zero return value terminates option processing, causing all later arguments to be positional (except possibly the following argument, if it is the value of the current one).

The characters of `o.sf` are the short-form letters, and `o.lf` is the long form; obviously, these should not both be empty. Unlike `optparse.py`, `optparse.c` does

[1] Probably `NULL`.

not require that the long form begin with two hyphens (or even one). After all, I compiled the example programs of §6 using `gcc -ansi -pedantic`. Note, however, that if a user misspells a long form starting with a single hyphen, a confusing error message will probably result since the argument will end up being parsed as a string of short-form options. For an option not taking a value, `o.metavar` should be empty; for one taking a value, it should be non-empty, and is used in forming the usage string for the option. For instance, if `o.sf` is `"f"`, `o.lf` is `"--file"` and `o.metavar` is `"FILE"`, the option names will be `-fFILE, --file=FILE`. The field `o.help` is the help text. If it is null, the option does not appear in the program help message; if it is `""` the option appears with empty help text. Any initial spaces are not printed with the help text, but at the start of the usage string, increasing its indentation.

To include text in the program help message that is not associated with an option, set `o.action` to `opt_text`. Then `o.sf` should be empty, and `o.data` has a special meaning: a non-null value specifies that, for the purposes of computing the starting position for option help text, the option list should be split into two just before this item. (See Example 2 in §6.) If `o.lf` is empty, `o.help` is filler; it is printed with indentation determined by any leading spaces. A non-empty value of `o.lf` is used to produce a usage string for a positional argument, which is formatted exactly as for an option (so `o.metavar` should be empty).

Macros.
The macros

```
OPT_NO_ACTION    OPT_NO_SF     OPT_NO_LF
OPT_NO_METAVAR   OPT_NO_HELP   OPT_NO_DATA
```

all expand to `(void *)0`. They may help you remember which fields of an `opt_spec` struct are being initialised to a null pointer when you read your code a week later.

3 Basic functions

The function opt_basename.
Synopsis. `void opt_basename(char *filename, char separator);`
Description. Modify `filename` in place to its base name, using `separator` as the path separator if it is non-zero, or `'/'` otherwise.

The function opt_config.
Synopsis.
```
void opt_config(int width, int max_help_position,
                int indent, const char *separator);
```
Description. Set parameters controlling the layout of the program help message. These all have default values; to use the defaults, given in brackets, pass a negative or zero value for `width` [79] or `max_help_position` [24], negative for `indent` [2], or an empty string for `separator` ["␣␣"]. The argument `width` gives the width in characters to be used for the help message, and `indent` gives the indentation for the option usage strings. Between the names of the option and its help text, `separator` is printed, padded on the left with spaces, if necessary, so that all help text starts

in the same column. If starting the help text on the same line as the option names would cause the help text to start to the right of `max_help_position`, the line is broken before the separator.

The function opt_options1st.
Synopsis. `void opt_options1st(void);`
Description. By default, options and positional arguments may appear in any order. If you call `opt_options1st` before `opt_parse`, the options must appear first; any argument after the first positional argument is treated as positional.[2]

The function opt_parse.
Synopsis.
```
int opt_parse(const char *usage,
              struct opt_spec *opts,
              char **argv);
```
Description. Parse the list of program arguments `argv`. The first program argument, `argv[0]`, is treated as the program name, and the arguments to be parsed are the remaining strings, terminated by a null pointer. (In help and error messages, `argv[0]` is used as is, so you may wish to call `opt_basename` on it first.) The argument `opts` should be an array of structs specifying the options to be recognized, terminated by one with a null pointer as its `action` field. The argument `usage` specifies the program usage string. It is used as the format string for a call to `fprintf`, with `argv[0]` as the single following argument. If it is empty, the default `"usage: %s [options]"` is used, which is fine for a program taking no positional arguments. By default, the next line of the help message is `options:`, but if the first element of `opts` has `opts[0].action` equal to `opt_text` and `opts[0].lf` empty, this is suppressed.
Side effect. Each non-positional argument has its first character replaced by 0.[3]
Special program arguments. The string `"--"` is treated as an option argument that terminates option parsing; all subsequent arguments are positional. The string `"-"` is a positional argument;[4] note that it doesn't terminate option parsing unless you've called `opt_options1st`.
Returns: `opt_parse` returns the number of positional arguments.

The function opt_remainder.
Synopsis. `char ***opt_remainder(void);`
Description. This should only be used within the action function of an option that takes (or can take) more than one value. The corresponding `metavar` field should be non-empty, so that `opt_parse` treats it as taking one value. Suppose the return value of `opt_remainder` has been assigned to a variable `p`. Then `*p` is the part of the `argv` array supplied to `opt_parse` that has not yet been examined. The effect of incrementing `*p` k times is to render k arguments invisible to `opt_parse`; note that

[2]Greg Ward, the author of `optparse.py`, believes that this behaviour is generally annoying to users. It is, however, used by Python, and seems quite natural for such a program.

[3]Yes, yes, it would be better if on exit the `argv` array contained only positional arguments, but the C standard does not guarantee that it is possible to do this.

[4]It can reasonably be argued that this isn't actually a special case, since it can't match any option.

your function is responsible for zeroing out those arguments. Here is the skeleton of an action function for an option requiring two values.

```
int gobble(char *val, void *data)
{
    char ***p = opt_remainder();
    if (!**p) opt_err("option %s requires two values");
    /* Do something with the value strings val and **p. */
    return **(*p)++ = 0;
}
```

Returns: `opt_remainder` returns a pointer to the variable used by `opt_parse` to step through the program arguments.

4 Error-reporting functions

The following functions should only be called from within the action function of an option. The typical reason for doing so is that the user gave an incorrect value for the option (say -5 for an option expecting a positive integer), but one might also call them if the user selected an option incompatible with an earlier one.

The function opt_name.
Synopsis. `const char *opt_name(void);`
Returns: `opt_name` returns the name of the option, in the form used in the argument, e.g. `"-f"` or `"--file"`, for use in error messages.

The function opt_err.
Synopsis. `void opt_err(const char *msg);`
Description. Print an error message determined by `msg` and exit. The body of this function is:

```
    opt_err_pfx();
    fprintf(stderr, msg, opt_name());
    opt_err_sfx();
```

If you want a message depending on run-time information other than `opt_name()`, you can replace the second line.

The function opt_err_pfx.
Synopsis. `void opt_err_pfx(void);`
Description. Print the prefix to an error message on `stderr`; currently, this is the program name followed by `":␣"`.

The function opt_err_sfx.
Synopsis. `void opt_err_sfx(void);`
Description. Print the suffix to an error message on `stderr` and exit. Currently, this prints `"\noption usage:\n"` followed by the usage string of the option in question.

5 Pre-defined actions

These are primarily intended to be stored in the `action` field of an `opt_spec` struct, but can also be called from within an action function you define. For each function, any constraints on its arguments are specified; after the declaration of the function in the synopsis there are up to two lines of code that could appear in the body of the function. One is an assertion that the first argument is null, or that it isn't null, which must be satisfied. The other assigns the second argument to a pointer to a specified type; the accesses through this pointer mentioned in the description must be valid. (Recall that when a function is stored as `o.action`, it will be called with arguments depending on `o.metavar` and `o.data`.)

The function opt_text.
Synopsis. `int opt_text(char *val, void *data);`
Description. This is a special case, whose only purpose is to be stored in an `action` field; it should never be called.

The function opt_help.
Synopsis.
```
int opt_help(char *val, void *data);
    assert(val == 0);
```
Description. Print the program help message and exit. If this function is stored as `o.action` and `o.help` is empty, `opt_parse` will use the default help text
`"print this help message and exit"`.
Returns: `opt_help` never returns.

The function opt_version.
Synopsis.
```
int opt_version(char *val, void *data);
    const char *p = data;
    assert(val == 0);
```
Description. Print the version string `p` and exit. If this function is stored as `o.action` and `o.help` is empty, `opt_parse` will use the default help text
`"print the version number and exit"`.
Returns: `opt_version` never returns.

The function opt_stop.
Synopsis. `int opt_stop(char *val, void *data);`
Description. This is the only pre-defined action that returns a non-zero value, halting option processing. Other than the return value, it acts like `opt_store_str` if `val` is non-null, and like `opt_store_1` if `val` is null and `data` is not, and the arguments must satisfy the constraints for the appropriate function. If both arguments are null, `opt_stop` does nothing but return a value.
Returns: `opt_stop` returns 1.

The function opt_store_0.
Synopsis.
```
int opt_store_0(char *val, void *data);
    int *p = data;
    assert(val == 0);
```
Description. Set `*p` to 0.
Returns: `opt_store_0` returns 0.

The function opt_store_1.
Synopsis. `int opt_store_1(char *val, void *data);`
```
int *p = data;
assert(val == 0);
```
Description. Set `*p` to 1.
Returns: `opt_store_1` returns 0.

The function opt_incr.
Synopsis. `int opt_incr(char *val, void *data);`
```
int *p = data;
assert(val == 0);
```
Description. Increment `*p`.
Returns: `opt_incr` returns 0.

The function opt_store_char.
Synopsis. `int opt_store_char(char *val, void *data);`
```
char *p = data;
assert(val != 0);
```
Description. If the string `val` has length 1, set `*p` to `*val`. Otherwise it is an error.
Returns: `opt_store_char` returns 0.

The function opt_store_int.
Synopsis. `int opt_store_int(char *val, void *data);`
```
int *p = data;
assert(val != 0);
```
Description. If the string `val` is the decimal representation of an integer, set `*p` to that integer. Otherwise it is an error.
Returns: `opt_store_int` returns 0.

The function opt_store_int_lim.
Synopsis. `int opt_store_int_lim(char *val, void *data);`
```
int *p = data;
assert(val != 0);
```
Description. The same as `opt_store_int`, except that it is an error if the integer is less than `p[1]` or greater than `p[2]`.
Returns: `opt_store_int_lim` returns 0.

The function opt_store_double.
Synopsis. `int opt_store_double(char *val, void *data);`
```
double *p = data;
assert(val != 0);
```
Description. If the string `val` represents a number, as specified for the standard C library function `strtod`, set `*p` to that number. Otherwise it is an error.
Returns: `opt_store_double` returns 0.

The function opt_store_double_lim.
Synopsis. `int opt_store_double_lim(char *val, void *data);`
```
double *p = data;
assert(val != 0);
```

Description. The same as `opt_store_double`, except that it is an error if the number is less than `p[1]` or greater than `p[2]`.
Returns: `opt_store_double_lim` returns 0.

The type struct opt_str and the function opt_store_str.
Synopsis.
```
struct opt_str {char *s, s0;};
int opt_store_str(char *val, void *data);
struct opt_str *p = data;
assert(val != 0);
```
Description. Set `p->s` to `val` and `p->s0` to `val[0]`. The reason for doing this is that, if `val` was a separate argument, after this function returns `opt_parse` will set `val[0]` to 0. (The reason for *that* is explained in footnote 3 on page 4.) The examples in §6 show how the calling program can cope sensibly with this situation.
Returns: `opt_store_str` returns 0.

The function opt_store_choice.
Synopsis.
```
int opt_store_choice(char *val, void *data);
const char **p = data;
assert(val != 0);
```
Description. Allows the user to choose one of the strings in the null-terminated array `p`. If the string `val` compares equal (using `strcmp`) to `p[i]` for some `i`, `p[0]` and `p[i]` are swapped. Otherwise it is an error, unless some `p[i]` is `""`, in which case this is swapped with `p[0]`. (In other words, if some `p[i]` is `""`, invalid choices are quietly ignored. Probably you'd write an action function that calls `opt_store_choice` and prints a warning if `""` has been "chosen".)
Returns: `opt_store_choice` returns 0.

The function opt_store_choice_abbr.
Synopsis.
```
int opt_store_choice_abbr(char *val, void *data);
const char **p = data;
assert(val != 0);
```
Description. The same as `opt_store_choice`, except that unambiguous abbreviations are accepted, and an ambiguous abbreviation is an unconditional error.
Returns: `opt_store_choice_abbr` returns 0.

6 Examples

The C optparse package contains two complete, if useless, programs, in the files `fudd.c` and `notpython.c`. Each program calls `opt_parse`, and if this returns prints which options were used to what effect, and any positional arguments remaining. Below I comment on some aspects of these programs.

Example 1. The program Fudd.

This is based on the example from the `optparse.py` documentation referred to earlier, and its help message is shown in Figure 1. Figures 2 and 3 show the whole program, with the first part setting up the options, and the second being the `main` function. Here are a few points to notice.

```
#include <stdio.h>
#include <stdlib.h>
#include "optparse.h"

int verbose = 1, group = 0;
struct opt_str fn = {NULL, 0};
const char *mode[] = {
    "intermediate", "novice", "expert", NULL
};

struct opt_spec options[] = {
    {opt_help, "h", "--help", OPT_NO_METAVAR, OPT_NO_HELP, OPT_NO_DATA},
    {opt_store_1, "v", "--verbose", OPT_NO_METAVAR,
     "make lots of noise [default]", &verbose},
    {opt_store_0, "q", "--quiet", OPT_NO_METAVAR,
     "be vewwy quiet (I'm hunting wabbits)", &verbose},
    {opt_store_str, "f", "--file", "FILE", "write output to FILE", &fn},
    {opt_store_choice, "m", "--mode", "MODE", "interaction mode: one of "
     "'novice', 'intermediate' [default], 'expert'", mode},
    {opt_text, OPT_NO_SF, OPT_NO_LF, OPT_NO_METAVAR,
     "  Dangerous Options:", OPT_NO_DATA},
    {opt_text, OPT_NO_SF, OPT_NO_LF, OPT_NO_METAVAR,
     "    Caution: use of these options is at your own risk.  "
     "It is believed that some of them bite.", OPT_NO_DATA},
    {opt_store_1, "g", OPT_NO_LF, OPT_NO_METAVAR, "  Group option.", &group},
    {OPT_NO_ACTION}
};
```

Figure 2: Setting up the options for Fudd.

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

    opt_basename(argv[0], '/');
    if (opt_parse("usage: %s [options] arg1 arg2", options, argv) != 2) {
        fprintf(stderr, "%s: 2 arguments required\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    printf("noise level: %s\n", verbose ? "verbose" : "quiet");
    if (fn.s) {
        fn.s[0] = fn.s0;
        printf("output file: %s\n", fn.s);
    }
    else
        puts("output file: none");
    printf("interaction mode: %s\n", mode[0]);
    printf("group option: %d\n", group);
    puts("positional arguments:");
    for (i = 1; i < argc; ++i) {
        if (*argv[i] && argv[i] != fn.s)
            printf("  %s\n", argv[i]);
    }
    return 0;
}
```

Figure 3: Fudd's `main` function.

- I decided that the user can repeat an option, with all but the last use being ignored, and can also use both the conflicting options `-v` and `-q`, with the last-used winning. Given this, we can use only pre-defined actions.
- In `options[0]`, we don't supply help text since the default is what we want.
- In the array `options`, the strings in the `help` fields of the 6th, 7th and 8th members have leading spaces that produce the same indentation in the help message as the use of an option group in `optparse.py`.
- After calling `opt_parse`, we make sure the value of `-f`, if supplied, has its first character correct:

  ```
  if (fn.s) {
      fn.s[0] = fn.s0;
      printf("output file: %s\n", fn.s);
  }
  ```

  Then, in printing the positional arguments, we have to check that a non-empty argument isn't the value of `-f`:

  ```
  for (i = 1; i < argc; ++i) {
      if (*argv[i] && argv[i] != fn.s)
          printf("  %s\n", argv[i]);
  }
  ```

Here is the output when several options other than `-h` are given.

```
leibniz:~/optparse> ./fudd string1 -qg -f outfile --mode=expert string2
noise level: quiet
output file: outfile
interaction mode: expert
group option: 1
positional arguments:
  string1
  string2
```

This shows an invalid value given to an option.

```
leibniz:~/optparse> ./fudd --mo movice
fudd: invalid choice 'movice' for option --mode
option usage:
  -mMODE, --mode=MODE  interaction mode: one of 'novice', 'intermediate'
                       [default], 'expert'
```

Example 2. The program NotPython.

The object of the exercise here is to write a program that accepts the same options, and produces the same help message, as Python. The help message of NotPython is shown in Figure 4, and you can check that it is the same as for Python 2.3.5 (except for the program name in the first line, of course). Because of the large number of options, the code is too long to reproduce here; see `notpython.c`. We

```
julia:~/optparse> ./notpython -h
usage: notpython [option] ... [-c cmd | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-c cmd : program passed in as string (terminates option list)
-d     : debug output from parser (also PYTHONDEBUG=x)
-E     : ignore environment variables (such as PYTHONPATH)
-h     : print this help message and exit
-i     : inspect interactively after running script, (also PYTHONINSPECT=x)
         and force prompts, even if stdin does not appear to be a terminal
-O     : optimize generated bytecode (a tad; also PYTHONOPTIMIZE=x)
-OO    : remove doc-strings in addition to the -O optimizations
-Q arg : division options: -Qold (default), -Qwarn, -Qwarnall, -Qnew
-S     : don't imply 'import site' on initialization
-t     : issue warnings about inconsistent tab usage (-tt: issue errors)
-u     : unbuffered binary stdout and stderr (also PYTHONUNBUFFERED=x)
         see man page for details on internal buffering relating to '-u'
-v     : verbose (trace import statements) (also PYTHONVERBOSE=x)
-V     : print the Python version number and exit
-W arg : warning control (arg is action:message:category:module:lineno)
-x     : skip first line of source, allowing use of non-Unix forms of #!cmd
file   : program read from script file
-      : program read from stdin (default; interactive mode if a tty)
arg ...: arguments passed to program in sys.argv[1:]
Other environment variables:
PYTHONSTARTUP: file executed on interactive startup (no default)
PYTHONPATH   : ':'-separated list of directories prefixed to the
               default module search path.  The result is sys.path.
PYTHONHOME   : alternate <prefix> directory (or <prefix>:<exec_prefix>).
               The default module search path uses <prefix>/pythonX.X.
PYTHONCASEOK : ignore case in 'import' statements (Windows).
```

Figure 4: The NotPython help message.

shall give those parts that seem noteworthy. The formatting of the help message is different from our default, and the options must come first (because Python has to pass arguments to the script it interprets, and these may look like Python options). Therefore just before the call to `opt_parse` are the lines

```
    opt_config(0, 0, 0, ": ");
    opt_options1st();
```

The first two arguments to `opt_config` select the default width and maximum help position, which are large enough not to cause unwanted line breaks. The line breaks in the long help strings don't correspond to word-wrapping at any width, so we put in explicit newlines. Before `main`, we define some variables used by action functions, one action function `storeW`, and a longish array `options` of `opt_spec` structs. The second line of the help message differs from the default, so the first element of `options` is

```
    {opt_text, OPT_NO_SF, OPT_NO_LF, OPT_NO_METAVAR,
     "Options and arguments (and corresponding environment variables):",
     OPT_NO_DATA},
```

There are 14 options; `-OO` is not a separate option, but represents giving the `-O` option twice, so the corresponding element of `options` pretends that `-OO` is a positional variable:

```
    {opt_text, OPT_NO_SF, "-OO", OPT_NO_METAVAR,
     "remove doc-strings in addition to the -O optimizations", OPT_NO_DATA},
```

Setting up the `-h` and `-V` options is easy:

```
    {opt_help, "h", OPT_NO_LF, OPT_NO_METAVAR, OPT_NO_HELP, OPT_NO_DATA},

    {opt_version, "V", OPT_NO_LF, OPT_NO_METAVAR,
     "print the Python version number and exit", "This is not Python 2.3.5"},
```

There are seven options that are simple switches (`-d`, `-E`, `-i`, `-S`, `-u`, `-v` and `-x`), so we define an array

```
int flags[7];
```

and each switch has an entry like

```
    {opt_store_1, "E", OPT_NO_LF, OPT_NO_METAVAR,
     "ignore environment variables (such as PYTHONPATH)", flags + 1},
```

The `-O` option has different effects depending on whether it is given once or twice, so we define a counter variable

```
int opt_level;
```

and an `options` element

```
    {opt_incr, "O", OPT_NO_LF, OPT_NO_METAVAR,
     "optimize generated bytecode (a tad; also PYTHONOPTIMIZE=x)",
     &opt_level},
```

The variable `opt_level` will end up holding the number of times `-O` was given. This being a toy program, we just print this number out; in real life, values greater than 2 could be treated as an error, or as equivalent to 2. The `-t` option is handled exactly like `-O`. The `-Q` option has a value which must be one of four strings. They must be entered exactly (I checked), so we define an array giving the choices

```
const char *divchoices[] = {"old", "warn", "warnall", "new", NULL};
```

and an element of `options`

```
    {opt_store_choice, "Q", OPT_NO_LF, " arg",
     "division options: -Qold (default), -Qwarn, -Qwarnall, -Qnew",
     divchoices},
```

The option `-c` takes a string value and also terminates the option list, which is handled by a variable

```
struct opt_str scriptstr;
```

and an item in `options`

```
    {opt_stop, "c", OPT_NO_LF, " cmd",
     "program passed in as string (terminates option list)", &scriptstr},
```

The remaining option, `-W`, is fairly complicated, and cannot be handled by a predefined action. Since this is just an example, it does not cope completely with this option. First, multiple `-W` options can be given. I set an arbitrary limit,

```
#define MAXW 10
```

and if this is exceeded print a warning message and ignore extra `-W` options. Next, the value of a `-W` option is a colon-separated list of up to 5 fields, of which the first is one of 6 (possibly abbreviated) strings (the action), and invalid values are ignored with a warning. In NotPython, I split the value into the first field and the tail. Then I call `opt_store_choice_abbr` to find out which of the 6 choices was supplied. If none of them was, a warning is given and the option ignored. Otherwise, I store the index of the chosen first field, along with the tail, which is not checked in any way. This is all implemented by: variables

```
int nW, Wactions[MAXW];
char *Wtails[MAXW];
```

to hold the number of `-W` options used, the indices of their first fields, and their tails; an array

```
const char *Wchoices[] = {
    "ignore", "default", "all", "module", "once", "error", "", NULL
};
```

giving the choices for the first field, together with `""` so that `opt_store_choice_abbr` will not treat an invalid choice as an error; the action function `storeW`, shown in Figure 5; and the element

```
    {storeW, "W", OPT_NO_LF, " arg",
     "warning control (arg is action:message:category:module:lineno)",
     OPT_NO_DATA},
int storeW(char *val, void *data)
{
    const char *first;
    char *tail;
    int i;

    if (nW == MAXW) {
        fprintf(stderr, "-W options after number %d are ignored.\n", MAXW);
        return 0;
    }
    first = Wchoices[0];
    tail = strchr(val, ':');
    if (tail)
        *tail = 0;
    opt_store_choice_abbr(val, Wchoices);
    for (i = 0; Wchoices[i] != first; ++i)
        ;
    Wchoices[i] = Wchoices[0];
    Wchoices[0] = first;
    if (Wchoices[i][0]) {
        Wactions[nW] = i;
        Wtails[nW++] = tail ? tail + 1 : "";
    }
    else
        fprintf(stderr, "Invalid -W option ignored: invalid action: '%s'\n",
                val);
    return 0;
}
```

Figure 5: The action function `storeW`.

of options.

Towards the end of options are three elements giving usage strings for positional variables, of which the first is

```
{opt_text, OPT_NO_SF, "file", OPT_NO_METAVAR,
 "program read from script file", OPT_NO_DATA},
```

Then comes a filler element:

```
{opt_text, OPT_NO_SF, OPT_NO_LF, OPT_NO_METAVAR,
 "Other environment variables:", ""},
```

Note the use of a non-null data field to cause the help text for the following items to start in a different column from that for the previous items. Finally we have four elements giving usage strings for environment variables, of which the first is

```
{opt_text, OPT_NO_SF, "PYTHONSTARTUP", OPT_NO_METAVAR,
 "file executed on interactive startup (no default)", OPT_NO_DATA},
```

The part of main after the call to opt_parse is straightforward.