```
/* coroutine.h
 *
 * Coroutine mechanics, implemented on top of standard ANSI C. See
 * https://www.chiark.greenend.org.uk/~sgtatham/coroutines.html for
 * a full discussion of the theory behind this.
 *
 * To use these macros to define a coroutine, you need to write a
 * function that looks something like this.
 *
 * [Simple version using static variables (scr macros)]
 * int ascending (void) {
 *    static int i;
 *
 *    scrBegin;
 *    for (i=0; i<10; i++) {
 *       scrReturn(i);
 *    }
 *    scrFinish(-1);
 * }
 *
 * [Re-entrant version using an explicit context structure (ccr macros)]
 * int ascending (ccrContParam) {
 *    ccrBeginContext;
 *    int i;
 *    ccrEndContext(foo);
 *
 *    ccrBegin(foo);
 *    for (foo->i=0; foo->i<10; foo->i++) {
 *       ccrReturn(foo->i);
 *    }
 *    ccrFinish(-1);
 * }
 *
 * In the static version, you need only surround the function body
 * with `scrBegin' and `scrFinish', and then you can do `scrReturn'
 * within the function and on the next call control will resume
 * just after the scrReturn statement. Any local variables you need
 * to be persistent across an `scrReturn' must be declared static.
 *
 * In the re-entrant version, you need to declare your persistent
 * variables between `ccrBeginContext' and `ccrEndContext'. These
 * will be members of a structure whose name you specify in the
 * parameter to `ccrEndContext'.
 *
 * The re-entrant macros will malloc() the state structure on first
 * call, and free() it when `ccrFinish' is reached. If you want to
 * abort in the middle, you can use `ccrStop' to free the state
 * structure immediately (equivalent to an explicit return() in a
 * caller-type routine).
 *
 * A coroutine returning void type may call `ccrReturnV',
 * `ccrFinishV' and `ccrStopV', or `scrReturnV', to avoid having to
 * specify an empty parameter to the ordinary return macros.
 *
 * Ground rules:
 *  - never put `ccrReturn' or `scrReturn' within an explicit `switch'.
 *  - never put two `ccrReturn' or `scrReturn' statements on the same
 *    source line.
 *
 * The caller of a static coroutine calls it just as if it were an
 * ordinary function:
 *
 * void main(void) {
 *    int i;
 *    do {
 *       i = ascending();
 *       printf("got number %d\n", i);
 *    } while (i != -1);
 * }
 *
 * The caller of a re-entrant coroutine must provide a context
 * variable:
 *
 * void main(void) {
 *    ccrContext z = 0;
 *    do {
 *       printf("got number %d\n", ascending (&z));
 *    } while (z);
 * }
 *
 * Note that the context variable is set back to zero when the
 * coroutine terminates (by crStop, or by control reaching
 * crFinish). This can make the re-entrant coroutines more useful
 * than the static ones, because you can tell when they have
 * finished.
 *
 * If you need to dispose of a crContext when it is non-zero (that
 * is, if you want to stop calling a coroutine without suffering a
 * memory leak), the caller should call `ccrAbort(ctx)' where `ctx'
 * is the context variable.
 *
 * This mechanism could have been better implemented using GNU C
 * and its ability to store pointers to labels, but sadly this is
 * not part of the ANSI C standard and so the mechanism is done by
 * case statements instead. That's why you can't put a crReturn()
 * inside a switch() statement.
 */
```

```
#ifndef COROUTINE_H
#define COROUTINE_H

#include <stdlib.h>

/*
 * `scr' macros for static coroutines.
 */

#define scrBegin         static int scrLine = 0; switch(scrLine) { case 0:;
#define scrFinish(z)     } return (z)
#define scrFinishV       } return
```

```
#define scrReturn(z)     \
        do {\
            scrLine=__LINE__;\
            return (z); case __LINE__:;\
        } while (0)
#define scrReturnV       \
        do {\
            scrLine=__LINE__;\
            return; case __LINE__:;\
        } while (0)

/*
 * `ccr' macros for re-entrant coroutines.
 */

#define ccrContParam     void **ccrParam

#define ccrBeginContext  struct ccrContextTag { int ccrLine
#define ccrEndContext(x) } *x = (struct ccrContextTag *)*ccrParam

#define ccrBegin(x)      if(!x) {x= *ccrParam=malloc(sizeof(*x)); x->ccrLine=0;}\
                         if (x) switch(x->ccrLine) { case 0:;
#define ccrFinish(z)     } free(*ccrParam); *ccrParam=0; return (z)
#define ccrFinishV       } free(*ccrParam); *ccrParam=0; return

#define ccrReturn(z)     \
        do {\
            ((struct ccrContextTag *)*ccrParam)->ccrLine=__LINE__;\
            return (z); case __LINE__:;\
        } while (0)
#define ccrReturnV       \
        do {\
            ((struct ccrContextTag *)*ccrParam)->ccrLine=__LINE__;\
            return; case __LINE__:;\
        } while (0)

#define ccrStop(z)       do{ free(*ccrParam); *ccrParam=0; return (z); }while(0)
#define ccrStopV         do{ free(*ccrParam); *ccrParam=0; return; }while(0)

#define ccrContext       void *
#define ccrAbort(ctx)    do { free (ctx); ctx = 0; } while (0)

#endif /* COROUTINE_H */
```