9.7. Non-local jumps

Provision is made for you to perform what is, in effect, a `goto` from one function to another. It isn't possible to do this by means of a `goto` and a label, since labels have only function scope. However, the macro `setjmp`and function `longjmp` provide an alternative, known as a *non-local goto*, or a *non-local jump*.

The file `<setjmp.h>` declares something called a `jmp_buf`, which is used by the cooperating macro and function to store the information necessary to make the jump. The declarations are as follows:

```
#include <setjmp.h>

int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

The `setjmp` macro is used to initialise the `jmp_buf` and returns zero on its initial call. The bizarre thing is that it returns **again**, later, with a non-zero value, when the corresponding`longjmp` call is made! The non-zero value is whatever value was supplied to the call of `longjmp`. This is best explained by way of an example:

```
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>

void func(void);
jmp_buf place;

main(){
        int retval;

        /*
         * First call returns 0,
         * a later longjmp will return non-zero.
         */
        if(setjmp(place) != 0){
                printf("Returned using longjmp\n");
                exit(EXIT_SUCCESS);
        }

        /*
         * This call will never return - it
         * 'jumps' back above.
         */
        func();
        printf("What! func returned!\n");
}

void
func(void){
      /*
       * Return to main.
       * Looks like a second return from setjmp,
       * returning 4!
       */
      longjmp(place, 4);
      printf("What! longjmp returned!\n");
}
```

Example 9.3

The `val` argument to `longjmp` is the value seen in the second and subsequent ‘returns’ from `setjmp`. It should normally be something other than 0; if you attempt to return 0 via `longjmp`, it will be changed to 1. It is therefore possible to tell whether the `setjmp` was called directly, or whether it was reached by calling `longjmp`.

If there has been no call to `setjmp`before calling `longjmp`, the effect of `longjmp` is undefined, almost certainly causing the program to crash. The`longjmp` function is never expected to return, in the normal sense, to the instructions immediately following the call. All accessible objects on ‘return’ from `setjmp` have the values that they had when `longjmp` was called, except for objects of automatic storage class that do not have volatile type; if they have been changed between the `setjmp` and `longjmp` calls, their values are indeterminate.

The `longjmp` function executes correctly in the contexts of interrupts, signals and any of their associated functions. If `longjmp` is invoked from a function called as a result of a signal arriving while handling another signal, the behaviour is undefined.

It's a serious error to `longjmp` to a function which is no longer active (i.e. it has already returned or another`longjump` call has transferred to a `setjmp` occurring earlier in a set of nested calls).

The Standard insists that, apart from appearing as the only expression in an expression statement, `setjmp` may only be used as the entire controlling expression in an `if`, `switch`, `do`, `while`, or `for` statement. A slight extension to that rule is that as long as it is the whole controlling expression (as above) the `setjmp` call may be the subject of the `!` operator, or may be directly compared with an integral constant expression using one of the relational or equality operators. No more complex expressions may be employed. Examples are:

```
setjmp(place);                    /* expression statement */
if(setjmp(place)) ...             /* whole controlling expression */
if(!setjmp(place)) ...            /* whole controlling expression */
if(setjmp(place) < 4) ...         /* whole controlling expression */
if(setjmp(place)<;4 && 1!=2) ...  /* forbidden */
```