Previous: Foreign Function Calls, Up: Foreign Function Interface

---

7.8 Step-By-Step Example of the Foreign Function Interface

This section presents a complete example of an interface to a somewhat complicated C function.

Suppose you have the following C function which you want to be able to call from Lisp in the file `test.c`

```
struct c_struct
{
  int x;
  char *s;
};

struct c_struct *c_function (i, s, r, a)
    int i;
    char *s;
    struct c_struct *r;
    int a[10];
{
  int j;
  struct c_struct *r2;

  printf("i = %d\n", i);
  printf("s = %s\n", s);
  printf("r->x = %d\n", r->x);
  printf("r->s = %s\n", r->s);
  for (j = 0; j < 10; j++) printf("a[%d] = %d.\n", j, a[j]);
  r2 = (struct c_struct *) malloc (sizeof(struct c_struct));
  r2->x = i + 5;
  r2->s = "a C string";
  return(r2);
};
```

It is possible to call this C function from Lisp using the file `test.lisp` containing

```
(cl:defpackage "TEST-C-CALL" (:use "CL" "SB-ALIEN" "SB-C-CALL"))
(cl:in-package "TEST-C-CALL")

;;; Define the record C-STRUCT in Lisp.
(define-alien-type nil
    (struct c-struct
            (x int)
            (s c-string)))

;;; Define the Lisp function interface to the C routine.  It returns a
;;; pointer to a record of type C-STRUCT.  It accepts four parameters:
;;; I, an int; S, a pointer to a string; R, a pointer to a C-STRUCT
;;; record; and A, a pointer to the array of 10 ints.
;;;
;;; The INLINE declaration eliminates some efficiency notes about heap
;;; allocation of alien values.
(declaim (inline c-function))
(define-alien-routine c-function
    (* (struct c-struct))
  (i int)
```

```
  (s c-string)
  (r (* (struct c-struct)))
  (a (array int 10)))

;;; a function which sets up the parameters to the C function and
;;; actually calls it
(defun call-cfun ()
  (with-alien ((ar (array int 10))
               (c-struct (struct c-struct)))
    (dotimes (i 10)                     ; Fill array.
      (setf (deref ar i) i))
    (setf (slot c-struct 'x) 20)
    (setf (slot c-struct 's) "a Lisp string")

    (with-alien ((res (* (struct c-struct))
                      (c-function 5 "another Lisp string" (addr c-struct) ar)))
      (format t "~&amp;back from C function~%")
      (multiple-value-prog1
          (values (slot res 'x)
                  (slot res 's))

        ;; Deallocate result. (after we are done referring to it:
        ;; "Pillage, *then* burn.")
        (free-alien res)))))
```

To execute the above example, it is necessary to compile the C routine, e.g.: `cc -c test.c && ld -shared -o test.so test.o' (In order to enable incremental loading with some linkers, you may need to say `cc -G 0 -c test.c')

Once the C code has been compiled, you can start up Lisp and load it in: `sbcl'. Lisp should start up with its normal prompt.

Within Lisp, compile the Lisp file. (This step can be done separately. You don't have to recompile every time.) `(compile-file "test.lisp")'

Within Lisp, load the foreign object file to define the necessary symbols: `(load-shared-object "test.so")'.

Now you can load the compiled Lisp ("fasl") file into Lisp: `(load "test.fasl")' And once the Lisp file is loaded, you can call the Lisp routine that sets up the parameters and calls the C function: `(test-c-call::call-cfun)'

The C routine should print the following information to standard output:

```
i = 5
s = another Lisp string
r->x = 20
r->s = a Lisp string
a[0] = 0.
a[1] = 1.
a[2] = 2.
a[3] = 3.
a[4] = 4.
a[5] = 5.
a[6] = 6.
a[7] = 7.
a[8] = 8.
```

```
a[9] = 9.
```

After return from the C function, the Lisp wrapper function should print the following output:

```
back from C function
```

And upon return from the Lisp wrapper function, before the next prompt is printed, the Lisp read-eval-print loop should print the following return values:

```
10
"a C string"
```