# E

# The C interface

## E.1 OVERVIEW

Eiffel software must interact with software written in other languages, particularly C and C++. With its emphasis on providing the right mechanisms for high-level system structuring, Eiffel is useful as a *wrapping* tool, which can serve to repackage existing software into better architectures based on object-oriented principles.

This chapter describes the possibilities available to ISE Eiffel users developer for calling external C functions. It is also possible to access and manipulate C++ classes; this is the topic of the next appendix.

The C interface provides in particular the following mechanisms, detailed in the following pages:

- You can specify that a certain external routine is in fact implemented on the C side as a macro, saving the overhead of function calls.
- You can force a certain type signature ("prototype") for the arguments and result of the C function in the Eiffel-generated C code.
- You can request certain include files to be included for a certain C function.
- You can call routines from Dynamically Linked Libraries (DLLs), in both their 16-bit and 32-bit variants on Windows. The library and the function can be specified explicitly in the Eiffel text, or they can be determined at run time; this last possibility is particular interesting if you are using the environment in "melt-only" mode, without a C compiler. *Windows*

These facilities go beyond what is described in *Eiffel: The Language* while remaining consistent with the language specification. (They simply rely on defining new external language names.) Among their various advantages, they enable Eiffel developers to make extensive use of external C code without having to write or maintain significant amounts of C code themselves — a desirable property since Eiffel developers tend to prefer working in Eiffel.

The discussion covers calling C from Eiffel. Some developments will also require interaction in the reverse direction: letting C/C++ code create Eiffel objects and apply Eiffel features to them. This important need is addressed by **Cecil**, the C-Eiffel Call-In Library. Cecil is described in chapter 24 of *Eiffel: The Language*; a presentation, with important additional information and examples, is available on-line from *http://www.eiffel.com*.

## E.2 EXTERNAL ROUTINES

The language specification given by the book *Eiffel: The Language* (called just *ETL* in the rest of this appendix) defines a simple way for Eiffel software to rely on other languages: a routine may be declared as **external**. The basic syntax is illustrated by the following example routine from *ETL*:

```
file_status (filedesc: INTEGER): INTEGER is
        -- Status indicator for filedesc
    external
        "C"
    alias
        "_ fstat"
    end
```

This declaration makes *file_status* usable in exactly the same way as any other Eiffel routine but implemented in a special way: as a call to a C function *_fstat*.

The **external** clause indicates that the routine's implementation is external; the string that follows is the name of the language, here C.

The **alias** clause indicates the name of the corresponding routine in the language of origin, here *_fstat*. You may omit the **alias** clause if the external name is the same as the Eiffel name; but in some cases a non-Eiffel routine might have a name which is not legal in Eiffel, as is the case here since a C function, unlike an Eiffel routine, may have a name beginning with an underscore _. We will also see how to use the **alias** clause to specify a function index in the case of a routine that encapsulates a C function from a Dynamically Linked Library.

The **external** mechanism requires the ability to generate C code, through freezing or finalizing, and hence requires the availability of a C compiler. It also requires you to indicate, in the Eiffel text, the name of the library, the name of the external routine, and its signature. Later in this appendix we will see a mechanism, DESC, that addresses both of these problems: the *SHARED_LIBRARY_ROUTINE* library class, through which you can call an external routine whose name is only determined at run time.

*See "DESC: CALLING A DLL ROUTINE DETERMINED AT RUN TIME", E.8, page 331.*

## E.3 EXTENDED LANGUAGE NAMES

When calling C functions, you can, as in the above example, choose the string *"C"* as language name. ISE Eiffel offers more possibilities, however: by adding qualifications after *C* you can specify further properties of the external functions. This only involves working on the contents of the "language name" string; since the language description in *Eiffel: The Language* does not place any restriction on the nature of that string, the facilities described in this chapter do not imply any departure from the official language specification.

The rest of this discussion describes the possible values of the string that follows the **external** keyword when the aim is to use C and C++ in various ways beyond the default mechanism. It shows first the syntax specification for External routine and continues with an explanation of the semantics followed by a few examples corresponding to common cases. The next appendix applies the same progression to the C++ interface.

*The examples are in "C EXAMPLES AND VALIDITY CONSTRAINTSS", E.6, page*

In both cases you may find it easier to look at the examples first.

# E.4 THE SYNTAX SPECIFICATION

In specification at the top of the next page, the syntax for External and External_name is retained unchanged from *ETL*, where Language_name is defined simply as Manifest_string; the rest covers the additional facilities..

| | | |
|---|---|---|
| External | $\triangleq$ | **external** Language_name [External_name] |
| Language_name | $\triangleq$ | '"'<br>Basic_language_name<br>[Special_external_declaration]<br>[Signature]<br>[Include_files]<br>'"' |
| Basic_language_name | $\triangleq$ | "C" \| "C++" |
| Special_external_declaration | $\triangleq$ | "[" Special_feature "]" |
| Special_feature | $\triangleq$ | Special_C_feature \| Special_C++_feature |
| Special_C++_feature | $\triangleq$ | ... See appendix F ... |
| Special_C_feature | $\triangleq$ | **macro** File_name \|<br>**dll16** File_name \|<br>**dll32** File_name |
| File_name | $\triangleq$ | User_file_name \| System_file_name |
| User_file_name | $\triangleq$ | '%"' Manifest string '%"' |
| System_file_name | $\triangleq$ | "<" Manifest_string ">" |
| Signature | $\triangleq$ | "(" Type_list ")" [Result_type] |
| Type_list | $\triangleq$ | {Type "," ...} |
| Result_type | $\triangleq$ | ":" Type |
| Include_files | $\triangleq$ | "\|" File_list |
| File_list | $\triangleq$ | {File_name "," ...} |
| External_name | $\triangleq$ | **alias** Manifest_string |

The syntax description conventions are those of *ETL*. $\triangleq$ means "is defined as". The vertical bar | separates alternative choices, such as the three possibilities for Special_feature. The brackets [...] enclose optional components; for example a Language_name is made of required quotes, a required Basic_language_name and three further components, any or all of which may be absent. The notation {*Element Separator* ...} means: 0 or more occurrences ("specimens") of *Element* separated, if more than one, by *Separator*.

Several of the symbols of the syntax notation, such as the brackets or the vertical bar, also appear in the language; to avoid any confusion they are given in double quotes. For example the specification for Include_files begins with "|" to indicate that an Include_files part starts with a vertical bar. Single quotes may be used instead of double quotes when one of the quoted characters is itself a double quote; for example '%"' appearing in the production for User_file_name denotes the percent character followed by the double quote character. Special words such as **macro** in bold italics stand for themselves, unquoted.

The whole Language_name part is a manifest string, beginning and ending with a double quote ". This is why any double quote character appearing in it, as in User_file_name, must be preceded, according to the Eiffel convention for strings, by a percent sign. As usual, a string that extends over two lines or more must have each line but the last end with a percent sign, and each last but the first start with a percent sign optionally preceded by any number of blanks and tabs.

## E.5 AVAILABLE POSSIBILITIES

A Special_C_feature, if present, indicates one of the following:

- If the special feature's declaration starts with **macro**, the external routine should be treated as a C macro, whose definition is to be found in the given File_name. This makes it possible to treat calls to the routine as simple macro expansions (even though from the Eiffel perspective they are seen as normal routine calls), avoiding the overhead of C function calls.

- If the declaration begins with **dll16** or **dll32**, the external routine belongs to a Windows Dynamically Linked Library, 16-bit in the first case and 32-bit in the second case. The compilation mechanism will take care of generating all the code that is necessary to prepare for interaction with the DLL routine. You must include a File_name, indicating where to find the DLL.

*Windows*

A Signature specifies the types that must be declared, in the generated C code, for the arguments and results. (The Result_type is optional; include it for a function but not for a procedure.) Normally the Eiffel compilation mechanism uses default conventions for the C equivalent for Eiffel types. These conventions, described at the end of this chapter, may occasionally be inadequate; for example a C function from an existing library may expect an argument of type *int16* (16-bit integer) rather than the default "long" integer type. By specifying the Signature explicitly, you can override the default conventions. Without this mechanism you would need to write a small C interface function which uses the default types and calls the existing function, "casting" the arguments (that is to say, forcing their conversion) into the required types. As noted at the beginning of this chapter, the aim of the facilities described here is to avoid the need for writing and maintaining such C code, and enable Eiffel developers to concentrate on writing Eiffel software even when they must develop interfaces with code written in other languages.

The Signature part, normally optional, is required in one case studied in more detail below: 32-bit DLLs with arguments.

*See "Using routines from Dynamically Linked Libraries", E.6.4, page 330.*

Finally, an Include_file part, if present, indicates that one or more files must be included, in the C sense, into the generated code. The include file may contain any C declarations that will be required by the external function.

A File_name is defined as either a User_file_name or a System_file_name. A User_file_name, in double quotes, is a normal path name; a System_file_name, in angle brackets, denotes an include file to be found in some fixed directory, according to conventions defined by the platform.

# E.6 C EXAMPLES AND VALIDITY CONSTRAINTSS

## E.6.1 Using a macro

Assume that **|path|usr|user_macro.h** contains the C macro definition

*Warning: this is C, not Eiffel.*

*#define sum(x,y)* ((*x*)+(*y*))

Then you can write the following Eiffel routine, whose calls will simply result in in-line expansion of the corresponding macro in the generated C code:

```
your_external (a, b: INTEGER): INTEGER is
    external
        "C [macro %"|path|usr|user_macro.h%"]"
    alias
        "sum"
    end
```

In this example and subsequent ones you should, as elsewhere in this manual, replace the vertical bar | by the path separator conventions of your platform.

## E.6.2 Casting the result

*For the Eiffel-C type equivalences see* *"EIFFEL-C TYPE EQUIVALENCE", E.7, page 331*.

To ensure that the result of a C function will be usable as an Eiffel *INTEGER*, even though it may be declared in C with another compatible type, you must cast it to the type *EIF_INTEGER* which is used, in the Eiffel-generated C code, to represent Eiffel integers. (*EIF_INTEGER* is declared by default as *long int*; the whole set of such types appears later in this chapter.) This is necessary to avoid type errors when C-compiling the generated C code. The Signature facility avoids the need to include any manual casting. Here is an illustration of how you can use it:

```
other_external (a, b: INTEGER): INTEGER is
    external
        "C [macro %"|path|usr|other_macro.h%"]%
            %(EIF_INTEGER, EIF_INTEGER): EIF_INTEGER"
    alias
        "sum"
    end
```

Note the two-line string, with the first line ending with a % and the second line starting with tabs and a %.

The example assumes that the C side returns a type other than *EIF_INTEGER*. The casting is only necessary for the result, since Eiffel *INTEGER* arguments will by default be declared in the generated C as *EIF_INTEGER*; but a Signature, if present, must include a type for every argument as well as the result.

If the C side required a type *your_type* (assumed to be compatible with *EIF_INTEGER*) for one of the arguments, you would use *your_type* at that position in the signature. In such a case *your_type* will often be a developer-defined C type rather than one of the predefined types of the language; the generated code must then have access to the type definition. This is a typical case in which the **include** facility, to be now explained, will prove handy.

### E.6.3 Specifying an include file

Here is what the preceding example becomes if we want the generated code to have access to two C include files, a system file and a user file:

```
your_external (a, b: INTEGER): INTEGER is
    external
        "C [macro %"|path|usr|other_macro.h%"]%
            %(EIF_INTEGER, EIF_INTEGER): EIF_INTEGER"%
            %| <|usr|include|xxx.h>, %"|path|user|other_macro.h%" "
    alias
        "sum"
    end
```

### E.6.4 Using routines from Dynamically Linked Libraries

Windows

The next two examples show how to use functions from Dynamically Linked Libraries on Windows. They require access to a C compiler; see below how to proceed without a C compiler (melt-only) or when the function name is not known at compile time.

Here is an Eiffel routine that encapsulates a function from a 16-bit DLL:

```
dynamic_external (a, b, c, d, e: INTEGER) is
    external
        "C [dll16 %"herlib16.dll%"] %
            % (WORD, DWORD, WORD, WORD, DWORD)"
    alias
        "67"
    end;
```

This example illustrates a property of the **dll16** and **dll32** subclauses: the use of a DLL index. To enable the generated code to call a routine from a DLL, you need to specify where to find the routine in the DLL. The external routine name (indicated in the **alias** clause if there is one, otherwise identical to the Eiffel routine name) serves this purpose. Instead of a name, however, the DLL mechanism may require an integer index; even if a name is accepted, providing an index is usually more efficient. To specify an index, write it in the **alias** clause as in the above example.

Note that specific C compilers for Windows may impose their own conventions, although 16-bit DLLs have reached a good level of interoperability, so that you can use the above scheme to call DLL routines generated by many popular C compilers.

For 32-bit DLLs there has not been as much *de facto* standardization yet. Our last example illustrates the use of a 32-bit DLL routine:

```
other_dynamic_external (a, b, c: INTEGER) is
    external
        "C [dll32 %"hislib32.dll%"] (WORD, DWORD, WORD)"
    alias
        "35"
    end
```

For DLL-32 routines the following two validity rules are enforced:

- An **alias** clause listing an integer is required.
- If the routine has arguments, the Signature part is required.

More generally, if you need to call 32-bit DLL routines produced with a specific C compiler, consult the release notes for ISE Eiffel as well as the documentation for the chosen C compiler.

### E.6.5 Sharing and freeing

If your system uses several routines from the same DLL, its execution will only load one instance of the DLL.

When the execution terminates, the ISE Eiffel run-time system will free all DLL instances loaded in this way.

## E.7 EIFFEL-C TYPE EQUIVALENCE

The type *EIF_INTEGER* was mentioned above. Here is the complete list of C type declarations used to declare the C equivalents of Eiffel's basic types:

*Warning: this is C, not Eiffel.*

```
typedef    long             EIF_INTEGER;
typedef    unsigned char    EIF_CHARACTER;
typedef    float            EIF_REAL;
typedef    double           EIF_DOUBLE;
typedef    char *           EIF_REFERENCE;
typedef    char *           EIF_POINTER;
typedef    unsigned char    EIF_BOOLEAN;
```

## E.8 DESC: CALLING A DLL ROUTINE DETERMINED AT RUN TIME *Windows*

All the mechanisms discussed so far for calling an external routine require that you include the routine's exact name in the Eiffel text (as the Eiffel routine name if it is the same, after **alias** otherwise). They assume, furthermore, that C code will be generated as the result of the compilation. Certain cases may invalidate either or both of these requirements:

- It may be desirable to wait until run time to determine the name of the external routine to be called in a DLL, or even the name of the DLL itself.
- You may use the environment without a C compiler, relying solely on melting.

The **Dynamic External Shared Call** mechanism (DESC for short) addresses both of these needs. Available on Windows, it is particularly important if you are using the environment without a C compiler, as it provides you with a mechanism for interfacing Eiffel with any outside tool as long as the interface can be set up through DLL routines.

In line with the general spirit of Eiffel, the DESC takes care of low-level aspects of DLL programming, freeing developers from operations which they would have to perform manually if they were using a language such as C: loading library instances; sharing these instances; freeing the instances when they are not needed any more.

Many platforms offer shared library facilities similar in spirit to the DLLs of Windows. Although they address the same general goals, these facilities vary widely in their details. The DESC mechanism is designed to provide a common framework enabling Eiffel developers to take advantage of library sharing in a consistent way across platforms. At present, however, it only supports the Windows variant, which we will now explore.

## E.8.1 DESC overview

The DESC mechanism enables you to construct objects representing external routines determined at execution time through their name and libraries, and to call these routines with the appropriate arguments.

Two classes, *DLL_16* and *DLL_16_ROUTINE*, supported by an auxiliary class *SHARED_LIBRARY_CONSTANTS* provide the basis for DESC under Windows:

- An instance of class *DLL_16* describes a 16-bit Dynamically Linked Library. This class is a descendant of the deferred class *SHARED_LIBRARY*, covering the platform-independent notion of shared library.
- An instance of class *DLL_16_ROUTINE* describes a routine from a 16-bit DLL. Predictably, this class has an attribute of type *DLL_16* describing the library to which the routine belongs. It has a deferred ancestor *SHARED_LIBRARY_ROUTINE* capturing the platform-independent notion of shared library routine.
- *SHARED_LIBRARY_CONSTANTS* introduces a few declarations useful for dealing with shared libraries and routines, in particular a few integer constants describing error codes and type codes. It is an ancestor to both of the preceding classes; application classes using DESC can also inherit from it to gain access to its facilities.

These classes are part of the Windows version of EiffelBase, available in precompiled form for use both with and without a C compiler.

The basic sequence of instructions to use the DESC mechanism is the following:

D1 • Create a library object (an instance of *DLL_16),* providing the library's name as argument to the creation procedure.

D2 • Create a routine object (an instance of *DLL_16_ROUTINE),* providing the library object, the routine's name or index in the library, and the routine's signature — number of arguments, types of arguments, type of result if any — as arguments to the creation procedure.

D3 • Apply procedure *call* to the routine object, passing to *call* an array that contains the actual arguments required by the external routine.

Each of these steps may be repeated as often as necessary to use multiple libraries, multiple routines in a library, or multiple calls to a given routine.

More details follow. (See also information in the Windows Release Notes, and the flat-short form of classes *DLL_16*, *DLL_16_ROUTINE* and *SHARED_LIBRARY_CONSTANTS*.)

## E.8.2 Creating a library object

To create a DESC object representing a library and load that library, use a declaration such as

*your_dll*: *DLL_16*

where *your_dll* should be replaced by whatever name you have chosen to denote the library in your software, and execute a creation instruction of the form

!! *your_dll*•*make* ("*your_lib_name*")

where *your_lib_name* is the name of the file containing the library.

After this call has been executed, the boolean value *your_dll*•*meaningful* will be true if and only if the creation has been successful, that is to say, the given name did correspond to an available library, and it was possible to load it.

If *your_dll*•*meaningful* is false, you can have more details about the error by comparing the value of *your_dll*•*error_code*, an integer, to those of constant attributes defined in class *SHARED_LIBRARY_CONSTANTS*. As expressed by an invariant of class *DLL_16*, the value of *meaningful* is true if and only if *error_code* = *0*.

## E.8.3 Creating a routine object

To create a DESC object representing a routine from a DLL, use a declaration such as

*your_routine*: *DLL_16_ROUTINE*

where *your_routine* should be replaced by whatever name you have chosen to denote the routine in your software, and execute a creation instruction of the form

```
!! your_routine.make_by_name
        (your_dll,
        "your_routine_name",
        <<argtyp1, argtyp2, ...>>,
        res_type)
```

or, if you prefer for faster access to identify the routine by an integer index rather than a name:

```
!! your_routine.make_by_index
        (your_dll,
        your_routine_index,          -- The only differing line
        <<argtyp1, argtyp2, ...>>,
        res_type)
```

where *your_dll* is the library object obtained at the previous step. The preconditions for both *make_by_name* and *make_by_index* include the following clauses on the first argument, known through its formal name *lib* (corresponding to *your_dll* above) in the routine:

```
require
    library_exists: lib /= Void
    meaningful: lib.meaningful
```

After either call has been executed, the boolean value *your_routine*•*meaningful* will be true if and only if the creation has been successful, that is to say, the given name or index did correspond to a routine of the library, and it was possible to open it. If the value is false, you can have more details about the error by comparing the value of *your_routine*•*error_code*, an integer, to those of constant attributes defined in class *SHARED_LIBRARY_CONSTANTS*. As expressed by a clause of the invariant of class *DLL_16_ROUTINE*, the value of *meaningful* is true only if *error_code = 0*.

Features *make_by_name* and *make_by_index* are usable not only as creation procedures but also as normal exported routines, so that you can later reinitialize the object to represent another external routine. The four arguments play the following roles:

- The first argument, as noted, denotes the library.
- The second argument identifies the desired routine in the library: by its name, of type *STRING*, in the first case; by its index, of type *INTEGER*, in the second.
- The third argument, of type *ARRAY* [*INTEGER*], gives the list of type codes for the arguments to the routine. Each type code is an integer associated with one of the possible types to be passed to a DLL routine.
- The fourth and last argument is a type code for the result.

In the above examples the third argument is declared as a manifest array; as explained in *Eiffel: The Language*, the notation ≪*a1*, *a2*, ...≫, known as a manifest array, denotes an array indexed from 1 whose successive items are the ones given. Here the array items *argtyp1*, *argtyp2*, ... must all be integers giving the type codes of the successive arguments to the routine. (Use an empty manifest array, ≪ ≫, if the routine has no arguments.) For these items as well as for *res_type* you may use the following type codes:

| | |
|---|---|
| *T_array* | Array. (What is passed to C is the "special object" containing the actual array elements, directly usable by C. To pass the Eiffel array object, use *T_reference*. A restriction: the elements of the array may be references, or they may be of a basic type — *BOOLEAN*, *INTEGER* etc. — but they may not be of an expanded type other than the basic types.) |
| *T_boolean* | Boolean (passed to C as unsigned character, 0 for false, nonzero for true). |
| *T_character* | Character. |
| *T_double* | Real number, double precision. |
| *T_integer* | Long integer type. |
| *T_real* | Real number, single precision. |
| *T_pointer* | Pointer to C structure. |
| *T_reference* | Reference to Eiffel object. |
| *T_string* | String. (What is passed to C is the C form of the Eiffel string; the conversion is achieved automatically by the feature *to_c* of class *STRING*. To pass the Eiffel string object, use *T_reference*.) |
| *T_short_integer* | Short integer type. (The Eiffel side will use normal *INTEGER* values for the corresponding actual arguments.) |
| *T_no_type* | No type; to be used for *res_type* only, for the case of a procedure (which has no result type). |

These type codes are constant integer attributes, whose declarations all appear in class *SHARED_LIBRARY_CONSTANTS*. The easiest way to use them, then, is to make sure that any class that uses the DESC mechanism inherits from *SHARED_LIBRARY_CONSTANTS*.

## E.8.4 Calling a routine

Having created the object representing the external routine and attached it to entity *your_routine*, you may now call the routine with arbitrary actual arguments through the procedure *call*, a feature of class *DLL_16_ROUTINE*.

The procedure takes a single argument, of type *ARRAY* [*ANY*], containing the successive actual arguments to be passed to the external routine. The easiest technique is to use a manifest array, as in

*your_routine*•*call* (<<–*325*, *67.2*, *x*, *a* + *b*>>)

## E.8.5 Accessing the result of a function

If *your_routine* denotes a function (a routine that returns a result), you will be able to access the result by querying the attached instance of *DLL_16_ROUTINE* through one of the following calls, each corresponding to one of the possible result types:

| Typical call | Eiffel type of the result |
|---|---|
| *your_routine*•*boolean_result* | *BOOLEAN* |
| *your_routine*•*character_result* | *CHARACTER* |
| *your_routine*•*double_result* | *DOUBLE* |
| *your_routine*•*integer_result* | *INTEGER* |
| *your_routine*•*pointer_result* | *POINTER* |
| *your_routine*•*real_result* | *REAL* |
| *your_routine*•*reference_result* | *ANY* (to use the result, an assignment attempt will usually be necessary) |
| *your_routine*•*string_result* | *STRING* (automatically converted from C form to Eiffel form by the feature *from_c* of class *STRING*) |

## E.8.6 Consistency requirements and protection against errors

In a call to procedure *call* such as the above, the number of elements in the array and their types must correspond to the signature — number and type of arguments — specified in the third argument of the latest call to *make_by_name* or *make_by_index*.

This requirement is captured by a function *conforms_to_signature*, relying on the function *conforms_to* from the Kernel Library class *GENERAL*. It is expressed by the third precondition clause of procedure *call*:

*call* (*args*: *ARRAY* [*ANY*]) **is**
    **require**
        *meaningful*: *meaningful*;

*valid_array*: *args* /= *Void*;
*conformant*: *conforms_to_signature* (*args*)

This precondition, combined with queries *meaningful* and *error_code* in classes *DLL_16* and *DLL_16_ROUTINE*, provides a certain degree of protection against possible errors. But the Eiffel side does not know anything about the external routine, and so cannot check that the number of actual arguments and their types match the actual signature of that routine. You are responsible for ensuring that the routine gets what it expects.

Similarly, each of the *_result* features has a precondition stating that it must be compatible with the result type set by the latest call to *make_by_name* or *make_by_index*. For example in the case of *boolean_result* the result type must have been set to *T_boolean*. Here too there is no protection against type errors at the Eiffel-C border; double-check your software to make sure that the result types you are positing on the Eiffel side match what the DLL routines actually declare.

## E.8.7 Sharing and freeing

As noted, one of the effects of creating a library object, using a creation instruction of the form !! *your_dll*•*make* ("*your_lib_name*"), is to load the library of name *your_lib_name*. When you subsequently create routine objects relative to *your_dll*, they will all share the same library instance.

It is possible, however, to load several instances of a library: simply create several library objects, passing in every case the same string "*your_lib_name*" as actual argument to the *make* creation procedure.

If the same library name is used by an external DLL routine, statically declared through the mechanism studied earlier in this chapter, and by a library object created dynamically by the DESC mechanism as an instance of *DLL_16*, two different instances will be loaded.

*See "Using routines from Dynamically Linked Libraries", E.6.4, page 330.*

When a DESC library object is no longer accessible and the garbage collector reclaims it, this will automatically (through the procedure *dispose* as redefined for class *DLL_16)* free the corresponding library instance.

For most uses this automatic freeing will be sufficient. If, however, you want to free a library manually, you can do so through the call *your_dll*•*free*. As a postcondition of this call, *your_dll*•*meaningful* will be false, as well as *your_routine*•*meaningful* for any routine object *your_routine* that was created relative to *your_dll*.