24

Interfaces with other languages

24.1 Overview

Eiffel software systems may need to use software elements written in other languages. This chapter describes the techniques which may be used to achieve such communication.

Many applications will be happy enough to use the pure Eiffel mechanisms described in the rest of this book, and will not require any direct interfaces with other languages. (The next section explains the circumstances which may justify the inclusion of foreign software in an Eiffel system.) If you are mostly interested in understanding the techniques of Eiffel proper, you should probably get familiar with the principles of external calls by reading this section and the next two, and move on to the next chapter.

Since Eiffel is of a higher level than common programming languages, and is particularly appropriate as an integration tool for combining software elements written in various languages, the more frequent need is for Eiffel routines to call non-Eiffel routines. The reverse need also exists, however. As a consequence, the mechanisms described in this chapter cover both *call-out* (Eiffel to foreign) and *call-in* (foreign to Eiffel) facilities. The call-in facilities depend heavily on the foreign language and, as a consequence, cannot be made part of the Eiffel description; they are presented in this chapter for illustration only.

In accordance with the terminology used for the different forms of Routine_body in the syntax specifications, this chapter will use the term **internal routine** for any Eiffel routine accessible to language processing tools, and **external routine** for other routines. The name “external” refers to the routine as viewed from the Eiffel text; for the form of the routine as it appears in its original language, the discussion will use the term **foreign routine**.

The notion of external routine, and the techniques described in this chapter, are clearly intended for routines written in other languages. True, if you have access to an Eiffel compiler, you could in principle precompile an Eiffel routine

and then have other routines call it as an an external routine. But there is no need to do this in a well-designed implementation of Eiffel: as discussed at the end of chapter 1, a good compiler will be incremental, avoiding any unnecessary recompilation of previously compiled routines; similarly, an interpreter should support calls to precompiled routines. In both cases, you will still benefit from type checking, in particular conformance checking of actual arguments against formal arguments. By treating Eiffel routines as external, you would lose these important validity checks.

The semantic specifications presented in this chapter involve the semantics of languages other than Eiffel. Granting non-Eiffel software access to Eiffel objects may defeat the properties guaranteed by the semantic rules of this book. You should exercise care, then, to guarantee that foreign languages are confined to their proper role in the construction of Eiffel-based software.

24.2 When to use external software

What is this proper role? After all, Eiffel is a complete programming language, and many systems do not need any external software.

Four cases, however, may require interfacing Eiffel classes with software written in other languages:

1 • Reuse of older software elements.

2 • Use of libraries written in other languages.

3 • Access to low-level platform-dependent properties.

4 • Use of Eiffel as a tool for re-engineering of software.

Cases 1 and 2 are similar: they result from the obvious observation that Eiffel developments do not proceed alone in the software world, but must be combined with other products. In case 1, an organization may want to reuse previously developed elements as part of a new system. In case 2, the system will use existing primitives providing facilities in a specialized area − such as graphics, databases, user interfaces, expert systems, or any other for which an external library is available.

In case 3, you need to access primitives which depend on the hardware or the operating system, available through external routines.

In case 4, an older non-Eiffel system must be converted to more modern software technology, but you want to proceed in stages. A possible strategy is to start by isolating appropriate abstractions in the existing software, and to build classes around them; the architecture of the resulting system will be expressed in Eiffel, using the structural mechanisms described in this book − classes, information hiding, genericity, inheritance, assertions − but the actual computations will still be performed by external routine calls. Here Eiffel serves as a packaging mechanism more than as a down-to-details programming language. This effort may be a first step towards more thorough re-engineering of the software: once you have an acceptable Eiffel structure, you may start upgrading to full Eiffel status some of the foreign elements in that structure, for example those which will need to evolve anyway because of changes in requirements, or those which are known to be the most difficult to maintain.

24.3 External routines

As seen in the discussion of routines, the Routine_body of an Effective routine, instead of using the more common Internal form (beginning with **do** or **once**), may be of the External form, which indicates that a call to the routine is a call to some outside software component.

← Routine_body *was discussed in 8.6, starting on page 113. The syntax is on page 113. The syntax for* External *appeared on page 406; it is reproduced below.*

An External clause begins with the keyword **external**, followed by a Manifest_string indicating the language in which the routine is written. It may also contain an External_name subclause, beginning with **alias**, giving the routine's name in its language of origin.

Here is an example of external routine:


```
f_close (filedesc: INTEGER): INTEGER is
        -- Close file associated with filedesc; record status in result.
    require
        descriptor_exists: exists (associated_file (filedesc));
    external
        "C"
    ensure
        (Result = 0) implies closed (associated_file (filedesc))
    end -- f_close
```

As this example shows, an external routine may have a Precondition and a Postcondition.

Function *f_close* performs a certain action and returns a status report through its result. This interface technique is not normally employed by Eiffel routines, which should instead record the status in an attribute; in communicating with external software, however, there may be no better means available.

You may wish to refer to an external routine through a name other than its original name in the foreign language. In such a case you may use an External_name subclause, beginning with **alias**, as in


```
file_status (filedesc: INTEGER): INTEGER is
    external
        "C"
    alias
        "_fstat"
    end -- file_status
```

The **alias** specifies that any call to *file_status* will cause a call to the C function of name *_fstat*. Two possible reasons may raise the need for such a subclause:

- The name adaptation is required for foreign routines whose native name is legal in the foreign language but not in Eiffel, as in the above example where a C function has a name beginning with an underscore character _, not acceptable as the first character of an Eiffel identifier. Similarly, other languages permit special characters such as *$* in identifiers.
- Even if the foreign name abides by Eiffel rules, it may be incompatible with the naming conventions that you have established for your project.

Here is the syntax of External routine bodies:

SYNTAX

External	≜	**external** Language_name [External_name]
Language_name	≜	Manifest_string
External_name	≜	**alias** Manifest_string

24.4 Executing an external call

SEMANTICS

The effect of a call to an external routine f was previewed in the general discussion of call semantics. Only three aspects differ from the semantics of Internal routines:

1 • Actual-formal argument association.

2 • Execution of the Routine_body

3 • Value to be returned, if the routine is a function.

The next section will cover items 1 and 3. Item 2, the simplest, was handled by the general discussion of call semantics. Quoting:

← *See 21.11, page 347, about feature application in a* Call*, and 21.14, page 352, for the precise specification of call semantics.*

> If df is an external routine, the effect of the call is to execute that routine on the actual arguments given, if any, according to the rules of the language in which it is written.

Here df is the version of f to be applied to the given target, deduced from the rules of call semantics (dynamic binding).

CAVEAT

In addition to its official arguments, an Eiffel routine has access to the **current object** − the target of the call. This important property does not necessarily hold for a foreign routine:

← *The notion of current object was explained on page 349.*

- If the foreign routine was written independently of Eiffel, it does not use the current object. Accordingly, the call, as specified by the above semantics, will not pass the current object. A typical case is a call to a primitive of a pre-existing graphics or database package.
- Another case is that of foreign routines specifically written for the needs of an Eiffel application. Such routines may need access to the current object; you must then explicitly pass *Current* as one of the arguments.

← *The entity 'Current' was introduced in 21.13, page 351.*

24.5 Argument and result transmission

The semantics of passing arguments, and of returning the result for a function, raises the problem of attachment between Eiffel values and foreign entities.

For internal routines, the semantic rule was simple, being deduced (like the semantics of Assignment instructions) from the semantics of the direct reattachment mechanism: at call time, each formal argument becomes attached to the corresponding actual; at return time, the result of a function is the final value attached to the function's *Result* entity.

← *See 21.11, page 347, and 21.14, page 352. The semantics of direct reattachment appears in 20.5, page 313.*

The semantic specification of a direct reattachment allowed flexible combinations of expanded and reference types in the source and target. Here is the table which gave the effect in all four possible cases:

SOURCE TYPE → / *TARGET TYPE* ↓	**Reference**	**Expanded**
Reference	[1] Reference reattachment	[3] Clone
Expanded	[2] Copy (will fail if source is void)	[4] Copy

← *This table appeared originally on page 317.*

This specification takes both types − source and target − into account, particularly in cases 2 and 3 where one is expanded and the other is not.

For external calls, however, we cannot afford such semantic flexibility, since the target is the formal argument, and we have no way of knowing how the foreign routine has declared it. The semantic definition must rely on properties of the actual argument alone.

LONG_ONLYTo depart as little as possible from the rules LONG_ONLYfor internal routines, the convention for external LONG_ONLYroutines is simple: follow the the semantics of direct reattachment, interpreted as if each formal argument were declared with **exactly** the same type as the corresponding actual.

This also applies to Current if it is one of the actual arguments: in accordance with the semantics of Current, defined by case D4 page 354, with an informal introduction on page 351, what is passed is a reference to the current object if the enclosing class is non-expanded, otherwise the current object itself.

This implies that only cases 1 and 4 of the above table make sense: either the actual argument is of a reference type, in which case the foreign routine will receive a reference, or it is of an expanded type, in which case the foreign routine will receive a copy of the attached object.

For the result of a function, the rule is similar: depending on the type declared for the function's result, the Eiffel side will expect the foreign routine to return a reference or an object.

Clearly, using foreign routines which will handle Eiffel values requires care. You must trust that the routine can manipulate the values it obtains from the Eiffel side, and, if it is a function, produces results which conform to what you expect. This implies that the types of arguments and result must be common to Eiffel and the external language.

For basic types, this property depends on both the foreign language and its implementation.

← *The basic types are BOOLEAN, CHARACTER, INTEGER, REAL, DOUBLE and POINTER. See chapter 32.*

For other types, no major problem will arise for a foreign routine which, given an object or reference, contents itself with a "store and forward": pass on the value to other routines, possibly keeping a copy in a variable of a suitable type. To do anything more with an Eiffel object, the routine must access its internal structure; it may avoid relying on implementation-dependent properties of object representation by using one of the following two portable mechanisms:

- The features of class *INTERNAL* from the Support Library provide access to the internal properties of objects (such as the various field values) with an implementation-independent interface.
- The Cecil library, described at the end of this chapter, offers access mechanisms for the specific case of the C language.

See "Eiffel: The Libraries" (reference in the bibliography of appendix C) about INTERNAL.

Remember that letting foreign routines operate on Eiffel objects may endanger the consistency of these objects; for example, it is easy to destroy an invariant. The semantic rules defined in this book do not take into account possible erroneous manipulations resulting from external calls. To obtain with certainty the semantics of Eiffel, you must use Eiffel routines only.

24.6 Passing the address of an Eiffel feature

In some cases a foreign routine may need to call Eiffel routines, or to access fields of Eiffel objects.

Foreign access to Eiffel routines may be necessary in particular for the implementation of so-called **call-back** mechanisms as they appear in such areas as user interfaces, graphics and databases. These are techniques enabling an application system to "plant" the address of one or more routines into another routine *r* at initialization time. Later, in various scheduled steps of its own algorithm, *r* will call the planted routines. Because planting is dynamic, the text of *r* does not show what actual routines will be called at the corresponding steps; it only contains "holes" where different applications may plant different routines. Often, *r* is a high-level loop, known as an **event loop**, which will repeatedly execute ritual actions (such as reading user input or updating the screen) provided by the planted routines.

In this description, you will have recognized the notion of **iterator** discussed in the presentation of inheritance and deferred features; indeed, the Eiffel techniques introduced for iterators, relying on deferred routines and dynamic binding, offer simpler, safer and more elegant alternatives to call-back.

← *On how to implement a call-back mechanism in Eiffel, see 10.13, page 149.*

In some cases, however, you may need to use an existing call-back mechanism implemented in another language, but with individual operations (the planted routines) provided by Eiffel routines. This means that you must be able to pass information about an Eiffel feature to a foreign routine *r*, enabling *r* to call that feature. For a routine, this information is the routine's address; the mechanism also applies to attributes.

The supporting construct is the Address form of Actual argument. An Address, introduced as part of the syntax for Actuals in the discussion of calls, is simply an actual argument of the form

← *The syntax for* Actuals *appeared on page 342.*

```
$ some_feature_identifier
```

This form of Actual argument is only useful for passing feature addresses to external routines. Internal routines do not need it, since the dynamic binding mechanism provides a better way to tell a supplier what routine it should call at a certain stage of the supplier's execution: you just pass the supplier an entity attached to a certain object; the dynamic type of that object, which may vary from one execution to the next, determines the applicable routine versions.

← *On dynamic binding, see 21.9 and 21.10, starting on page 345.*

One might expect a constraint prohibiting Actual arguments of the Address form except in calls to external routines. There is no such constraint, however, because it may be useful for an Internal routine *ir1* to pass the address of a routine *r* to another internal routine *ir2*, so that *ir2* may itself pass *r* to an external routine *er*. Were this not permitted, *ir1* would need to call *er* directly, which may not be the desired scheme.

← *The hypothetical constraint, an addition to condition 4 of the argument validity rule of page 369, would require 'df' to be external.*

We must prevent *ir2* from performing any operation on its argument *r* other than passing it along to another routine. A type rule achieves this:

An argument of the Address form is of type *POINTER*.

As a consequence, the declaration for the corresponding formal argument in the receiving routine (whether Internal or External) must be of the form

ir2 (*eiffel_routine*: *POINTER*; ...) **is** ...

POINTER, a Kernel Library class, serves only for the purpose discussed here: as type for arguments of the Address form. It has no exported feature; so the body of *ir2*, if Internal, may not apply any feature to *eiffel_routine*.

← *The argument validity rule appears on page 369.*

The validity constraint on actual arguments of the Address form is clause 4 of the argument validity rule, which makes *$ f* valid as actual argument to a call if and only if *f* is the final name of a feature of the enclosing class, and that feature is not a constant attribute (which has no address).

If the rule is satisfied, the feature will have a version *df* applicable to the current object: this is the version of *f* for the current object's generator (taking into account possible renaming and redefinition). The value passed for attachment to the corresponding formal argument is the address of *df*. This applies to both routines and variable attributes; for an attribute, the call will pass the address of the field corresponding to *df* in the current object.

Clearly, this is meaningful only if the foreign language has a way of dealing with pointers to values and routines.

If *df* is a routine, foreign software components will be able to call that routine. Such calls require one extra argument, appearing at the first position and corresponding to the target of the call. Assume for example a routine

some_routine (*a1*: *A*; *b1*: *B*) **is** ...

Calls to *some_routine* in Eiffel texts may be qualified or unqualified:

target.*some_routine* (*x, y*)
some_routine (*u, v*)

Assume now that a call to an external routine *ext* makes the address of *some_routine* available to a foreign language:

ext (..., *$ some_routine*, ...)

Let *sr* be the formal argument for *some_routine* in the foreign routine corresponding to *ext*. The foreign routine will call *some_routine* with one extra actual argument, appearing at the first position:

sr (*target, x, y*)
sr (*current_object, x, y*)

WARNING: *calls to 'sr', appearing here in Eiffel-like syntax, will in practice be written in a foreign language, which may use different notations for calls and for denoting a routine of address 'sr'.*

The extra argument denotes the call's target, which in Eiffel appeared before the dot (as in the case of *target*) or not at all (as with *current_object*). It denotes an object or object reference, passed to the foreign routine as argument to an external call, or obtained through one of the call-in facilities described next.

24.7 The Cecil Library

The just explored Address form of actual argument provides a **call-in** mechanism, enabling foreign software to call Eiffel features, and complementing the call-out facilities studied earlier. It requires the Eiffel side to pass the needed feature addresses as arguments to external calls.

Some developers may wish to write foreign routines which create Eiffel objects and apply features to these objects, without relying on features explicitly passed by the Eiffel side. This last section shows a way to do this from C, using a library of C functions called the C-Eiffel Call-In Library, or Cecil. It is not hard to transpose the Cecil techniques to other foreign languages.

Most developments do not need to use Cecil or its equivalent, and most developers do not need to learn about it. The ideas are of interest to installations with a heavy use of C or some other foreign language, if they want to integrate some Eiffel classes in applications driven by their foreign components. If you are not in this situation, then you most likely should spare yourself the rest of this chapter; but do shed a tear or two for your less fortunate colleagues.

Please send your tax-deductible contributions to the HAVOC fund (Help All Victims Of C!), Box OO, Palma de Majorca.

Call-in mechanisms belong in foreign languages. What this section describes, then, is not part of Eiffel. The NICE Consortium may eventually decide to standardize a call-in facility; Cecil will then be an obvious candidate. This section describes a library for interfacing with C software. The library is not yet a part of Eiffel although we hope it will form the basis for the standard in this area.

The Cecil library contains macros, functions, types and error codes. All have names beginning with either *eif_* (functions and macros) or *EIF_* (types and error codes); examples are the function *eif_type_id* and the type *EIF_PROC*, explained below. Their declarations appear in a C "header file", *cecil.h*, which you may add to a C program through the C preprocessor directive

Eiffel's concern for clarity suggests prefixes 'eiffel_' and 'EIFFEL_', but the resulting names would be too long for some C compilers.

```
#include <cecil.h>
```

WARNING: this is C, not Eiffel.

Let us now look at the principal facilities declared in *cecil.h*. First of all, the C side will need to refer to Eiffel types. It will know a type through a "type-id", of type *EIF_TYPE_ID*. To obtain a type-id for a non-generic class of name *CLASSNAME* and record it in a C variable *your_id*, use the function *eif_type_id*, which returns a result of type *EIF_TYPE_ID*:

```
EIF_TYPE_ID your_id;
...
your_id = eif_type_id ("CLASSNAME");
```

WARNING: this is C, not Eiffel.

If the class is generic, replace the last instruction by

```
your_id = eif_generic_id ("CLASSNAME", gen1, gen2, ...);
```

WARNING: this is C, not Eiffel.

where *gen1, gen2, ...* are type-ids corresponding to the desired actual generic parameters. Function *eif_generic_id* has a variable number of arguments; the number of arguments following the first one ("*CLASSNAME*") must match the number of formal generic parameters of the class of name *CLASSNAME*.

The CLASSNAME is the "external" name of the class, which may be different from its Eiffel name as directed by an Ace's Visible *part. See below.*

The result returned by *eif_type_id* or *eif_generic_id* describes a type which is expanded if and only if *CLASSNAME* is declared in Eiffel as **expanded class**. To force a result describing an expanded type, apply *eif_expanded* to the result of either function; the result is another type-id. All these functions return as result the error code *EIF_NO_TYPE* if they cannot compute a type-id (no class with the given name in the universe, more than one class, wrong number of generic parameters).

In the presence of an optimizing compiler which may remove "dead code", you must make sure that the required class is not optimized away. If your implementation supports Lace, the Visible part of the corresponding cluster specification serves this purpose; an External_class_rename subclause will also remove any ambiguities if two or more visible classes have the same name.

→ *See appendix D about Lace, in particular D.14, page 535, about* External_class_rename.

A C function may access Eiffel objects through references passed to it by the Eiffel side in external calls, or returned by calls to *eif_create* (see below). The corresponding variable must be declared of the Cecil type *EIF_OBJ*.

A value *your_object* of type *EIF_OBJ* is **not** necessarily a C pointer to the corresponding object. To obtain such a pointer (for example to pass it to a C function which manipulates objects directly), use the macro *eif_access*, which takes an *EIF_OBJ* and returns a pointer, of type *char∗* . For example:

Current C ideology suggests that 'eif_access' should return a 'void∗ '. But many C compilers only accept 'char∗ '.
WARNING: this is C, not Eiffel. The result is a null pointer if 'your_object' corresponds to a void reference.

```
some_function (eif_access (your_object), ...);
```

The reason for this rule is that an Eiffel implementation supporting garbage collection may move objects around. Then a pointer passed directly to a C function might be obsolete by the time the function tries to access the associated object. Given an *EIF_OBJ*, *eif_access* will retrieve a correct pointer. If the implementation does not move objects, *eif_access* will do little or no work.

What if *your_object* is a variable which does not just allow immediate object processing as above, but retains its value between successive activations of the C side? In the meantime, the Eiffel side might have discarded all references to the corresponding object; but then a garbage collecting implementation must not be allowed to reclaim it! To avoid this, the C side must **adopt** the object, using the function *eif_adopt*. Once C functions do not need to hold the object any more, they may release it through *eif_wean*. Here is the scheme:

WARNING: this is C, not Eiffel.

```
EIF_OBJ your_object; ...
your_object = eif_adopt (your_object);

   ... Then in the same or another C program unit: ...
some_function (eif_access (your_object), ...);
...
eif_wean (your_object);
```

As announced above, it is possible to create an object from C, using the function *eif_create* which takes an *EIF_TYPE_ID* argument and returns an *EIF_OBJ*. For example:


```
EIF_OBJ your_list;
...
your_list = eif_create (eif_generic_id ("LINKED_LIST",
                        eif_type_id ("INTEGER")));
```

WARNING: this is C, not Eiffel. A C programmer may prefer to include the initialization of 'your_list' in its declaration.

Assuming class *LINKED_LIST* with one generic parameter, this creates a direct instance of *LINKED_LIST* [*INTEGER*]. Function *eif_create* calls *eif_adopt*; the C side should call *eif_wean* when and if it does not need the object any more.

As you will have noticed, *eif_create* **does not call a creation procedure**. To apply a creation procedure, you will need to include a separate call, using function *eif_proc* as explained below. This departs from Eiffel conventions, which prohibit creating an object without applying a creation procedure if the class has a Creators clause. With Cecil, forgetting to call a creation procedure after *eif_create* may produce an object which violates the class invariant, so you must be particularly vigilant to avoid this error (which is impossible in Eiffel).

← *About creation rules in Eiffel and the* Creators *clause, see chapter 18.*

Next, the C side will want to apply Eiffel routines to objects. To do so it needs C pointers to these routines, which it will obtain through one of a set of Cecil functions provided for this purpose. For example, having obtained the type-id *your_list* as shown above, use the following to assign to variable *your_procname* a pointer to a procedure whose Eiffel name in class *LINKED_LIST* is *go*:


```
EIF_PROC your_list_go;
...
your_list_go = eif_proc ("go", your_id);
```

WARNING: this is C, not Eiffel. A C programmer may prefer to include the initialization of 'your_list_go' in its declaration.

Function *eif_proc*, returning a result of type *EIF_PROC*, is one of a group of functions, each corresponding to a different category of Eiffel routines: procedures, functions returning results of basic types, functions returning references to complex objects. (The case of Bit_type is treated below.) Here is the list of functions used for calling the various kinds of Eiffel features from C, with their types and template arguments:

```
EIF_PROC eif_proc (routine_name, type_id)
EIF_FN_BOOL eif_fn_bool (routine_name, type_id)
EIF_FN_CHAR eif_fn_char (routine_name, type_id)
EIF_FN_INT eif_fn_int (routine_name, type_id)
EIF_FN_REAL eif_fn_real (routine_name, type_id)
EIF_FN_DOUBLE eif_fn_double (routine_name, type_id)
EIF_FN_POINTER eif_fn_pointer (routine_name, type_id)
EIF_FN_REF eif_fn_ref (routine_name, type_id)
```

The word POINTER in the name EIF_FN_POINTER refers to the Eiffel basic type (used to pass feature addresses to external routines, see 24.6 above), not to C pointers.

In all cases the arguments are a string, representing a routine name, and a type-id (obtained through *eif_type_id* or *eif_generic_ic*):

```
char * routine_name; EIF_TYPE_ID type_id;
```

WARNING: this is C, not Eiffel.

These functions look for a routine of name *routine_name* in the base class of the type corresponding to *type_id*. If no such routine exists, the result is a null pointer. Otherwise it is a pointer to a C function representing the desired routine; you may then call that function on appropriate arguments. For example:


```
(your_list_go) (eif_access (your_list), 10)
```

WARNING: this is C, not Eiffel. You may use '(∗ your_list_go)' instead of '(your_list_go)'.

This applies the routine corresponding to *go*, accessible through *your_list_go* as a result of the above call to *eif_proc*, to the object corresponding to *your_list*, with the actual argument *10*. The corresponding call would have been written in Eiffel as *your_list.go* (*10*). In C, do not forget to enclose the name of the function pointer, here *your_list_go*, in parentheses, and to use *eif_access*.

There is a major difference between the Eiffel call and its C emulation: Cecil **does not apply dynamic binding**. What you get from *eif_proc* or one of its sisters is a pointer to a function representing the exact Eiffel routine of the given name in the given class. In the presence of polymorphism and redeclaration, the Eiffel call may trigger a different version of *go* depending on the type of the object attached to *your_list* at the time the call is executed. The C form will always call the same version, regardless of the object's type.

Not using dynamic binding is of course potentially dangerous; but then computations which need the benefits of object-oriented technology should be written in Eiffel, not C. Cecil is useful for a C call to an Eiffel routine when you can statically specify, through a routine name and a class name, the exact computation that you require.

The next Cecil facility enables the C side to access fields of complex objects, corresponding to attributes of the generating classes. To obtain a field of an object, use the macro *eif_field*. For example:


```
EIF_OBJ your_object, your_other_object; int your_field_copy;
...
your_field_copy = eif_field (eif_access (your_object),
                             "some_integer_attribute", EIF_INTEGER);
eif_field (eif_access (your_object), "some_complex_attribute",
                   EIF_OBJ) = your_other_object;
```

WARNING: this is C, not Eiffel.

As shown by this example, you may use the result of *eif_field* in two different ways: as an expression, or "r-value" in C terminology; or as a writable entity, or "l-value", which may then be the target of an assignment. Such an assignment will re-attach the corresponding object field. The two instructions illustrate these two cases: the first assigns the result of the call to *your_field_copy*; the second changes the value of the field *some_reference_attribute* in *your_object*.

Function *eif_field* takes three arguments. The first is a value of type *EIF_OBJ*, representing an object; do not forget to protect it by *eif_access*. The second is a string giving the name of the desired attribute. The third is one of the following values, describing the type of the attribute:

EIF_BOOLEAN	*EIF_CHARACTER*	*EIF_INTEGER*
EIF_REAL	*EIF_DOUBLE*	*EIF_POINTER*
EIF_REFERENCE		

These names are those of macros defined in 'cecil.h'.

The result of *eif_field* is undefined if the object does not have a field with the given name and type.

As with routines, you may have to guard against a compiler unduly optimizing the attribute away, and you may need to use as attribute name (second argument to the above functions) an external name different from the name appearing in the class text. If your implementation supports Lace, the Visible part of the corresponding Ace will take care of these aspects.

→ *See D.14, page 535, about* Visible *parts in Aces.*

There remains the case of functions and attributes of a Bit_type. Its treatment is in line with the techniques just described for other types, but requires a special convention because C has no direct equivalent to the Eiffel notion of Bit_type. A Bit_type is expanded; so for example the possible values for an entity of type *BITS 256* are sequences of 256 bits. In C this is not possible; the values of the corresponding variables will be **pointers** to such bit sequences.

To call an Eiffel function returning a Bit_type result, use this scheme:

← Bit_type *and the corresponding semantics were defined in 12.14, page 209.*

```
EIF_BIT your_bit;
EIF_FN_BIT your_function;
...
your_function = eif_fn_bit (function_name, type_id);
your_bit = (your_function) (eif_access (object), actual_1, actual_2, ...)
```

WARNING: *this is C, not Eiffel. A C programmer may prefer to include the initialization of 'your_function' in its declaration.*

The last instruction assigns to *your_bit* a reference to the bit sequence returned by the function. *EIF_BIT* describes a pointer type.

Similarly, you may access and modify Bit_type fields as follows:

```
EIF_BIT your_bit1, your_bit2;
...
your_bit1 = eif_bit_field (eif_access (your_object), "some_bit_attribute");
eif_bit_set_field (eif_access (your_object), "some_bit_attribute", your_bit2);
```

WARNING: *this is C, not Eiffel.*

Here two primitives are needed. Function *eif_bit_field* takes two arguments, an object pointer (returned by *eif_access*) and an attribute name. In contrast with *eif_field*, the result of *eif_bit_field* may only be used as an expression (r-value), not as a writable variable. To change the value of a Bit_type field, use *eif_bit_set_field*, whose last argument is the new value, of type *EIF_BIT*. If the object has no appropriate Bit_type field, *eif_bit_field* returns the value *EIF_NO_BFIELD* and the effect of *eif_bit_set_field* is undefined.

The following primitives are applicable to *your_bit* of type *EIF_BIT*:

```
eif_bit_set (your_bit, position)
eif_bit_clear (your_bit, position)
eif_bit_ith (your_bit, position)
eif_bit_length (your_bit)
eif_bit_clone (your_bit)
```

Here *position* is an integer index which must be in the range of the bit sequence (counted from 1); otherwise the first two calls have undefined results, and *eif_bit_ith* returns *EIF_NO_BIT*. Routines *eif_bit_set* and *eif_bit_clear* set the element of index *position* to 1 and 0, respectively. Function *eif_bit_ith* returns an integer, the value of the element of index *position*. Function *eif_bit_length* returns an integer, the length of the sequence. Function *eif_bit_clone* returns an *EIF_BIT*, a fresh copy of the bit sequence passed as argument.