31

Interfacing with C, C++ and other environments

31.1 OVERVIEW: THE COMPONENT COMBINATOR

Object technology as realized in Eiffel is about **combining components**. Not all of these components are necessarily written in the same language; in particular, as organizations move to Eiffel, they will want to reuse their existing investment in components from other languages, and make their Eiffel systems interoperate with non-Eiffel software.

Eiffel is a "pure" O-O language, not a hybrid between object principles and earlier approaches such as C, and at the same time an **open** framework for combining software written in various languages. These two properties might appear contradictory, as if consistent use of object technology meant closing oneself off from the rest of the programming world. But it's exactly the reverse: a hybrid approach, trying to be O-O as well as something completely different, cannot succeed at both since the concepts are too distant. Eiffel instead strives, by providing a coherent object framework — with such principles as Uniform Access, Command-Query Separation, Single Choice, Open-Closed and Design by Contract — to be a *component combinator* capable of assembling software bricks of many different kinds.

The following presentation describes how Eiffel systems can integrate components from other languages and environments.

The more frequent case of external interfaces is *call-out*: Eiffel routines calling non-Eiffel ones. The reverse need (foreign to Eiffel, or *call-in*) also exists. The mechanisms described in this chapter cover both.

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 what circumstances may including 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 four, and move on to the next chapter.

If you do study the details, you will note that they include, particularly in the specific external sublanguages supporting interaction with C, C++ and Dynamic Link Libraries, a number of specific mechanisms that may appear too rich when compared to the general sobriety of Eiffel's design. Do not be put off by this wealth of possibilities; the aim is not to complicate Eiffel but to enable Eiffel developers to take full advantage of non-Eiffel software at minimum effort. Any new, advanced technology such as Eiffel must provide effective bridges to older technologies, so that its users can leverage off existing investment. In particular, having powerful C and C++ interface sublanguages won't detract you from the simplicity of Eiffel programming; the effect instead will be that if you *do* have to interface with C and C++ you will be able to do everything you need on the Eiffel side, rather than having to write special "*glue code*" in those languages. Eiffel programmers, remarkably, prefer to program in Eiffel; carefully crafted interface sublanguages enable them to talk freely to the rest of the world without having to leave their language, techniques and tools of choice.

"Even under extreme duress, 99.9873% of Eiffel programmers still choose Eiffel", in Proc. of STOOP-SOLOW (joint meeting of Society for Torturing Object-Oriented Programmers and Society for Observing the Limits of Object Work), Sing-Sing (NY), Jan. 2001, pp. 5670-8782.

In accordance with the terminology used for the different forms of Routine_body in the syntax specifications, the discussion 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; the form of the routine as it appears in its original language will be called the **foreign routine**.

→ In special cases the "other" language might be Eiffel itself. See below.

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 to confine the foreign languages to their proper role, avoiding unwanted interference with Eiffel object structures and algorithms.

31.2 WHAT EIFFEL CAN DO WITH THE REST OF THE WORLD

Here is some of what you can do with the foreign language facilities described in this chapter.

- You may declare an Eiffel routine as **external**, specifying that it comes from a foreign language. To the rest of the Eiffel software, the routine looks as if it were a normal Eiffel routine; but calls to it will execute the foreign code, which must of course have been compiled by a compiler for the foreign language. This is possible in principle for any foreign language, and guaranteed for C, C++, Java and Fortran 95.
- You may specify that an external routine, known in Eiffel under a certain name, had **another name** in its native language, for example if that name is not legal in Eiffel.
- You may specify that an external routine is actually implemented by a C **macro**, avoiding the overhead of function calls.

- You may associate a function and a procedure — a “getter” and a “setter” — to a C structure (“**struct**”), so that a call to the function will automatically access, and a call to the procedure modify, a specified field of that structure.
- You may even **include C code inline** in the body of an external routine, so that the external routine is in this case “internal” in the sense that it is specified within the Eiffel code, rather than elsewhere.
- You may use from Eiffel the routines of a **DLL** (Dynamic Link Library). You may specify the library and routines in your Eiffel text or, to make the process fully dynamic, you may obtain or compute this information at run time, just when you need to access the DLL elements.
- You may use from Eiffel all the facilities of a C++ class: **member functions**, **static members**, **data members**, **constructors**, **destructors**.
- You may use the *Legacy++* tool to produce a **C++ class wrapper**: an Eiffel class, automatically generated, that makes *all* the facilities of a C++ class (as listed above: member functions, data members and so on) available to the rest of the Eiffel system.
- Going the other way around, you may use the *Cecil* library to let external software do everything with an Eiffel system that you can do in Eiffel: create Eiffel objects, call on them any of the features of the corresponding classes, and so on. In other words Cecil lets you treat an Eiffel system as a **package** that the rest of the world can use as a library.
- That library can be dynamic: you can **generate a DLL** from an Eiffel system.
- You can also generate **COM components** (for Microsoft’s Component Object Model) and even XYZ components for execution on the XYZ virtual machine.

The next sections describe these mechanisms in detail, after a brief review of the proper role of foreign software elements in the development process.

31.3 WHEN TO USE EXTERNAL SOFTWARE

Why use external software? 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.

Both cases 1 and 2 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 — graphics, databases, user interfaces, expert systems...

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, encompassing the internals as well as the structure. This is not an all-or-nothing decision: you may redo some of the components in Eiffel, for example the most advanced or innovative ones, and leave some others in the original language if they are stable and satisfactory.

The external facilities, detailed in the rest of this chapter, include:

- The possibility of specifying a routine as External, to indicate that it is written in another language and compiled separately; this notion will occupy the major part of the discussion.
- As a special case of the External mechanism, the C-Eiffel Interface Sublanguage, and the corresponding C++ facilities, enabling Eiffel software to take advantage of special foreign facilities such as C's macros and C++'s constructors (next section).
- The **Legacy++** tool for automatic Eiffel wrapping of C++ classes.
- **Cecil**, the C-Eiffel Call-In Library, allowing other languages to use almost all of Eiffel's facilities. (The initial C is in the acronym for historical reasons, but Cecil can be used from any other language.)

31.4 REGISTERED LANGUAGES AND THE ROLE OF C

Eiffel's external facilities depend in part — especially in the call-in case — on the properties of external languages; short of covering every programming language in existence, the specification cannot be exhaustive. It includes explicit knowledge about a few languages, said to be the **registered languages**, currently C, C++, Java, Fortran 95 and Eiffel itself. Any Eiffel compiler must support an interface to the registered languages, as described in this chapter.

Including Eiffel among the registered "foreign" languages is more a matter of completeness than of obvious necessity. Although in principle this allows you to integrate previously compiled Eiffel classes as if they were external software,better way are usually available; a good Eiffel environment should be able to treat such classes like other Eiffel classes and perform all the relevant type checking. Another possible use of Eiffel as registered foreign language is to integrate Eiffel classes compiled with another compiler, although better interoperability mechanisms are desirable.

Among the registered languages, **C**, and its more recent variant **C++**, play a particular role for a number of technical, political and historical reasons:

- Since the mid-nineteen seventies, C has become the low-level *lingua franca* of computing, available on almost all platforms and known to a growing majority of programmers.
- Almost all dominant operating systems are written in C sometimes with more recent additions in C++.
- Most programs — from operating systems and database management systems to graphical libraries, object request brokers and other component-based development tools, development environments and many others — provide an Application Programming Interface (API) for C programs if they provide an API at all. When they offer more than one API, the one for C is often the reference. So a carefully engineered C binding is critical for many industrial developments.
- C compilers have benefited from wide use and several decades of research on compilation technology, aimed at producing efficient code.
- Although C has undergone changes, source code portability is reasonably good for programmers who follow some basic precautions.
- Many Eiffel implementations, such as ISE Eiffel, compile to C, taking advantage of the preceding properties, in particular wide availability, portability, and efficient code generation.
- A high-level language, Eiffel needs a good intermediary to access facilities from the machine and the operating system. C, more effective as a tool for use by *programs* than by humans, plays that role quite well. Libraries such as EiffelBase go to C when they occasionally must get out of the high-level language framework to access the nuts and bolts of the machine. C then plays for the Eiffel programmer exactly the same role that assembly language plays for the C programmer.

For all these reasons a special set of facilities — almost a mini-language within Eiffel, the *C-Eiffel Interface Sublanguage* — is available for those programmers who need fine-tuned access to C mechanisms from Eiffel. The Sublanguage allows you for example to use C macros, "structs", include files, C dynamic link libraries (DLLs), or even to include *inline C code* in Eiffel routines.

→ "THE C INTERFACE SUBLANGUAGE", 31.11, page 832.

Similar possibilities also exist for C++, giving Eiffel access to the components of C++ classes — member functions, constructors, destructors — and complemented by the automatic Legacy++ wrapper.

→ "THE C++ INTERFACE SUBLANGUAGE", 31.12, page 837.

The role of these facilities is quite clear: to take the best advantage of C software, while writing *as little C as possible*. Eiffel programmers prefer writing Eiffel. They know that the world isn't all Chanel perfumes and candlelight dinners, and that once in a while one must tender to the more mundane necessities of life. But then they expect the Eiffel compiler, through the Eiffel-C interface, to do much of the grunt work, and limit their use of C to the indispensable minimum.

31.5 BASICS OF EXTERNAL ROUTINES

We now start the study of the basic foreign affairs construct, External.

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 8.5, page 218. The syntax is on page 218. The syntax for External appeared on page 819; 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 (or, in the case of inline C routines, the actual C text).

Here is an example of external routine

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

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 technique is not normally employed by Eiffel functions, which should instead record the status in an attribute; in communicating with external software, however, there may be no better way.

You may use an External_name subclause, beginning with **alias**, to refer to an external routine through a name other than the one it has in the foreign language. For example:

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

The **alias** specifies that any call to *file_status* will cause a call to the C function of name *_fstat*. There are two possible reasons for such a subclause:

- The native name may be legal in the foreign language but not in Eiffel, as in the *file_status* example where the function name *_fstat*, legal in C, is illegal in Eiffel since it starts with an underscore.
- Even if the foreign name abides by Eiffel rules, it may violate the naming conventions of your project.

In the absence of an **alias** subclause, the feature name passed to the external software is the **lower name** of the feature.

← *The lower name is the name all in upper case. See “TEXTUAL CONVENTIONS”, 2.13, page 101.*

So even if you give to an external feature a name following the letter case conventions of another language, such as *SetValue* for an external routine implemented in C, the name passed to C will be *setvalue*. Even if it is implemented as an external routine, an Eiffel feature should follow Eiffel conventions: call it *set_value* and use **alias** "*SetValue*".

Here is the basic syntax of External routine bodies:

External routines

```
              External ≜ external External_language [External_name]

     External_language ≜ Unregistered_language |
                         Registered_language

 Unregistered_language ≜ Manifest_string

         External_name ≜ alias Manifest_string
```

The External clause is the mechanism that enables Eiffel to interface with other environments and serve as a “component combinator” for software reuse and particularly for taking advantage of legacy code.

By default the mechanism assumes that the external routine has the same name as the Eiffel routine. If this is not the case, use an External_name of the form **alias** "ext_name". The name appears as a Manifest_string, in quotes, not an identifier, because external languages may have different naming conventions; for example an underscore may begin a feature name in C but not in Eiffel, and some languages are case-sensitive for identifiers whereas Eiffel is not.

Instead of calling a pre-existing foreign routine, it is possible to include **inline** C or C++ code; the **alias** clause will host that code, which can access Eiffel objects through the arguments of the external routine.

The language name (External_language) can be an Unregistered_language: a string in quotes such as "**Cobol**". Since the content of the string is arbitrary, there is no guarantee that a particular Eiffel environment will support the corresponding language interface. This is the reason for the other variant, Registered_language: every Eiffel compiler must support the language names "**C**", "**C++**" and **dll**. Details of the specific mechanisms for every such Registered_language appear below.

Some of the *validity* rules below include a provision, unheard of in other parts of the language specification, allowing Eiffel language processing tools to rely on *non-Eiffel tools* to enforce some conditions. A typical example is a rule that requires an external name to denote a suitable foreign function; often, this can only be ascertained by a compiler for the foreign language. Such rules should be part of the specification, but we can’t impose their enforcement on an Eiffel compiler without asking it also to become a compiler of C, C++ etc.; hence this special tolerance.

The general *semantics* of executing external calls appeared as part of the general semantics of calls. The semantic rules of the present discussion address specific cases, in particular inline C and C++.

Although you may intermix routines of the External and Internal forms, it is common practice to separate the two categories, grouping external routines into their own Feature_clause. In some cases you will even find “wrapper” classes consisting mostly or entirely of external routines, encapsulating a set of external facilities into an abstraction usable directly by the rest of the Eiffel software.

31.6 EXECUTING AN EXTERNAL CALL

Before exploring the varieties of foreign interfacing mechanisms, we must understand the precise semantics of external calls, previewed in the general discussion of call semantics. Only three aspects differ from the semantics of Internal routines:

1 • Actual-formal argument association.

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

3 • Execution of the Routine_body

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

← *“PRECISE CALL SEMANTICS”, 23.17, page 643*.

> 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).

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

← *The notion of current object was defined on page 641*.

• 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.

← *“Current object and routine”, page 640*.

31.7 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.

← *“PRECISE CALL SEMANTICS”, 23.17, page 643. The semantics of direct reattachment was in 22.7, page 585*.

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 → *TARGET* ↓	**Reference**	**Expanded**
Reference	[1] Reference reattachment	[3] Clone
Expanded	[2] Copy (fails if source void)	[4] Copy

← This table originally appeared on page 588.

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.

To depart as little as possible from the rules for internal routines, the convention for external routines, follow 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: with the semantics of Current, defined by case 2, page 644, 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. So 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 (chapter 30) are BOOLEAN, CHARACTER, INTEGER, REAL, their sized variants and POINTER.

For other types, no major problem will arise for a foreign routine which, given an object or reference, just needs to do 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 EiffelBase 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, allows foreign languages to access Eiffel features.

31.8 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 **callback** mechanisms as they appear in such areas as user interfaces, graphics and databases. Callback enables routines to “plant” the address of one or more routines into another routine *r* at initialization time. Later, at various places in 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) through 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. But you may need to use an existing call-back mechanism implemented in another language, with individual planted operations to be provided by Eiffel features. So you need the ability to pass to an external routine the address of an Eiffel feature.

← On how to implement a call-back mechanism in Eiffel, see 10.15, page 271.

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 618.

$ *feature_or_parenthesized_expression*

Here *feature_or_parenthesized_expression* can be the name of an Eiffel feature, a parenthesized expression such as (*a* + *b*), as well as *Current* or, in a function, *Result*. In all cases what is passed is an address. For a feature this enables the foreign software to call the feature; for an expression it gives it access to a location containing the value of the expression. The latter is useful for a foreign routine that expects not a value but an address containing that value.

This Address form of Actual argument is only useful for passing such addresses to external routines. Internal (Eiffel) routines do not need it, since the dynamic binding mechanism provides a better way to tell a supplier what feature 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 23.12 and 23.13, starting on page 630.

Here is the syntax for an Address argument:

Address ≜ "**$**" Address_mark

Address_mark ≜ Variable

Feature_name is the most common case.

As to the validity constraint, we saw it as part of the Argument rule, which makes *$ f* valid as actual argument to a call if and only if *f*, when an Extended_feature_name, is the final name of a feature of the class.

← Page 626.

An Address argument, as noted, describes the address of a routine or expression. It is subject to a constraint:

Address rule *VZAR*

An Address is valid if and only if its Address_mark is of a reference type.

An expanded type would not make sense here as its values have copy rather than reference semantics.

How do we describe an "address" in Eiffel? A basic type is available for that purpose: *POINTER*, described by a Kernel Library class. Hence the type rule:

Address Type rule

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

As a consequence, the declaration for the corresponding formal argument in the receiving routine must be of the form

```
ir2 (...; from_eiffel: POINTER; ...) is ...
```

or the corresponding declaration in a foreign language.

Note that this routine can indeed be an Internal Eiffel routine as well as an external one. Although you might expect Address actual arguments to be permitted only in calls to external routines, there is no such constraint: 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 the argument validity rule of page 626, would require the called routine df to be external.

We must prevent *ir2* from performing any operation on its argument *r* other than passing it along to another routine. This simply follows from the properties of class *POINTER*, which has no exported features except for the universal, harmless features *copy*, *clone*, *equal* and consorts from *ANY*. So all you can do on an argument of type *POINTER* — other than copying it, cloning it, comparing for equality and so on — is to pass it on to someone else.

Address semantics

The value of an Address expression is a *POINTER* enabling foreign software to access the associated Variable.

The manipulations that the foreign software can perform on the corresponding pointer depend on the foreign programming language. It is the implementation's responsibility to ensure that such manipulations do not violate Eiffel semantic properties.

--- REWRITE (MOSTLY REMOVE) THE REST OF THIS SECTION ----

Now the semantics of an Address argument **$** *f* being passed to a routine *r*. We must distinguish between the possible cases for *f*:

1 • If *f* is an Extended_feature_name (as noted, the most common case), the corresponding feature have a version *df* applicable to the current object, taking into account possible renaming and redefinition. *df* is the feature that a call *x*.*f* (...) would execute, according to the rules of dynamic binding, when *x* is attached to an object of the current type. The value passed to *r* 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 useful only if the foreign language can deal with addresses of fields and routines.

2 • If *f* is a constant attribute or a Parenthesized expression, what is passed to the routine is the address of a memory location containing its value.

3 • If *f* is *Current*, the value passed is the address of the current object.

4 • If *f* is *Result*, the value passed is the address used to store the result to be returned by the enclosing function.

In case 1, where *f* denotes a feature, foreign software elements will be able to call that feature. Such calls require one extra argument, appearing at the first position and corresponding to the target of the call. Assume

```
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 (x, y)
```

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)
```

These calls to sr appear here in Eiffel syntax, but the convention for calls in the foreign language may be different.

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.

The above calls to *sr* from a foreign language are examples of what what the beginning of this chapter defined as the **call-in** case: exercising Eiffel mechanisms from the outside. To take this scheme to its full realization the foreign software needs:

- A way to manipulate Eiffel objects safely (protecting them, in particular, from the Eiffel garbage collector).
- A clear correspondence between the types of Eiffel and those of the foreign language.
- An adequate calling mechanism for features.

The Cecil library, described later in this chapter, provides all of this. But we are not ready yet to move on to call-in facilities, since we are not finished with call-out. In addition to the language-independent call-out constructs just studied, Eiffel's external interface offers special support for C and C++ — languages important enough to deserve mini-sublanguages of their own in the Eiffel syntax for External features.

→ "THE CECIL LIBRARY", 31.16, page 855.

31.9 SPECIAL INTERFACE SUBLANGUAGES

We saw that the syntax for declaring a routine as External involves a language name:

← This syntax appeared first on page 819.

External languages

External ≜ **external** External_language [External_name]

External_language ≜ Unregistered_language | Registered_language

Unregistered_language ≜ Manifest_string

External_name ≜ **alias** Manifest_string

The External_language may be an Unregistered_language — a plain Manifest_string describing an arbitrary language; this is useful only if that language is known to your specific Eiffel compiler, or uses default argument passing conventions that will work with Eiffel. But it may also be a Registered_language, covering DLL routines, which may come from any language, and the four languages guaranteed to be handled properly:

Registered languages

Registered_language ≜ C_external | C++_external | DLL_external

IL_external refers to the Intermediate Language of the Microsoft .NET framework.

The cases of , C_external, C++_external and DLL_external give rise to special sublanguages with a host of detailed possibilities, reviewed in the next three sections. Note that all the C possibilities are also available for C++, so in practice the third sublanguage is a superset of the second.

31.10 GENERAL SUBLANGUAGE MECHANISMS

The specific sublanguages — C_external, C++_external and DLL_external — offer common techniques for specifying certain elements:

- Routine signatures.
- Files needed to use the external software, for example C include files or the files containing a DLL.
- Types used to establish a precise correspondence between the type systems of Eiffel and those of other languages (for example, between an Eiffel *INTEGER* and a C *int*).

Before going into the specific sublanguages, let us review these shared facilities in turn.

Specifying an external routine signature

Since external languages have their own type systems, you may need to specify that a certain routine expects certain types for its arguments. In languages such as C and C++ that support "casts" (forced conversions), these types will be used for casting the arguments.

To specify types in the relevant sublanguages you may include an External_signature in the string specifying the language, as in the C external function declaration

```
your_external (a, b: INTEGER): INTEGER
        external
                "C signature (int, int)"
        end
```

The External_signature part in this example is

```
(int, EIF_INTEGER_32)
```

indicating that the associated C function expects two arguments of the C type *int* (integer). The names listed must be types of the external language, such as *int* for a C routine. *EIF_INTEGER_32* is a type used for the correspondence between Eiffel and C types, as explained in a later section.

→ "Controlling the Eiffel-C type correspondence", page 836.

It doesn't matter that *int* and *EIF_INTEGER_32* are not valid Eiffel type names: remember that an External_signature such as the above, like everything else in the sublanguages under discussion, appears in a string.

As you will have noted, the External_signature only lists types for arguments; for a function, you cannot specify a type, because the compiler will make sure that the function's result is converted back to the result type specified for the Eiffel routine. (In this respect the construct name External_signature and the keyword **signature** are a little misleading, since elsewhere in the description of Eiffel the word "signature" covers both result and argument types, but it still seems to be the best name here.)

The syntax of External_signature is straightforward:

External signatures

External_signature ≜ **signature** [External_argument_types] [**:** External_type]

External_argument_types ≜ "**(**" External_type_list "**)**"

External_type_list ≜ {External_type "**,**" …}*

External_type ≜ Identifier

The External_signature, if at all present, must cover all arguments:

External Signature rule *VZES*

An External_signature in the declaration of an external routine *r* is valid if and only if it satisfies the following conditions:

1 • Its External_type_list contains the same number of elements as *r* has formal arguments.

2 • The final optional component (**:** External_type) if present if and only if *r* is a function.

A language processing tool may delegate enforcement of these requirements to non-Eiffel tools on the chosen platform.

The rule does not prescribe any particular relationship between the argument and result types declared for the Eiffel routine and the names appearing in the External_type_list and the final External_type if any, since the precise correspondence depends on foreign language properties beyond the scope of Eiffel rules.

→ On this correspondence in the C case, see “Controlling the Eiffel-C type correspondence”, page 836.

The specification of a non-external routine never includes C-style empty parenthesization: for a declaration or call of a routine without arguments you write *r*, not *r* (). The syntax of External_argument_types, however, permits **()** for compatibility with other languages’ conventions.

The last part of the rule allows Eiffel tools to rely on non-Eiffel tools if it is not possible, from within Eiffel, to check the properties of external routines. This provision also applies to several of the following rules.

External signature semantics

An External_signature specifies that the associated external routine:

- Expects arguments of number and types as given by the External_argument_types if present, and no arguments otherwise.
- Returns a result of the External_type appearing after the colon, if present, and otherwise no result.

Specifying external files

To use an external routine, you may need to provide one or more file names:

- A C or C++ function may rely on some "include files"; for example, the type *EIF_INTEGER_32* used by *your_example* above must have a C definition, to which the C function must have access. It will find it in an include file, which you may specify from the Eiffel side.
- To use an external routine from a DLL, you must indicate the file that contains the DLL.

An External_file_use part, starting with **use**, enables you to say which files you need. Here is its application to the preceding example, assuming you want function *your_external* to have access to two C include files:

```
your_external (a, b: INTEGER): INTEGER
        external "[
                C
                        signature (int, int)
                        use <stdio.h>, "/path/user/her_include.h"
                ]"
        end
```

This example and several that follow use a multi-line Verbatim_string, written between an opening **"[** and a closing **%"**. We could also use a plain string without this convention, but then the internal double quote signs ", in the specification of the path name, would have to be written %"; also, interrupted lines would need to finish with a %, and continuation lines to start with a %.

← *"MANIFEST STRINGS", 29.8, page 784*.

Here is the syntax of External_file_use:

External file use

External_file_use ≜ **use** External_file_list

External_file_list ≜ {External_file "," …}*

External_file $\triangleq$ External_user_file | External_system_file

External_user_file $\triangleq$ ' **"** ' Simple_string ' **"** '

External_system_file $\triangleq$ "<"Simple_string ">"

As the syntax indicates, you may specify as many external files as you like, preceded by **use** and separated by commas. You may specify two kinds of files:

- "System" files, used only in a C context, appear between angle brackets < > and refer to specific locations in the C library installation.
- The name of a "user" file appears between double quotes, as in "*/path/user/her_include.h*", and will be passed on literally to the operating system. Do not forget, when using double quotes, that this is all part of an Eiffel Manifest_string: you must either code them as %" or, more conveniently, write the string as a Verbatim_string, the first line preceded by **"[** and the last line followed by **]"**.

An External_file refers to file and path names. Different operating systems have different conventions to denote paths; to avoid worrying about these differences, the examples of this chapter assume the Unix/Linux style using forward slash characters, as in */path/usr/file.c*. This convention is also understood by most C compilers on Windows, even though the native Windows style uses backslash characters, as in *d:\path\usr\file.c*. VMS has its own notation.

The difference between the two forms of External_file is that a C_user_file, of the form "*path_name*", denotes a file through its exact location in the file system, whereas a C_system_file of the form "<*file_name*>" is relative to the location of standard include files — such as stdio.h for standard C input and output — in the C installation.

In either case, any files listed must exist and have the expected contents:

External File rule *VZEF*

An External_file is valid if and only if its Simple_string satisfies the following conditions:

1 • When interpreted as a file name according to the conventions of the underlying platform, it denotes a file.

2 • The file is accessible for reading.

3 • The file's content satisfies the rules of the applicable foreign language.

A language processing tool may delegate enforcement of these conditions to non-Eiffel tools on the chosen platform.

Condition 3 means for example that if you pass an include file to a C function the content must be C code suitable for inclusion by a C "include" directive. Such a requirement may be beyond the competence of an Eiffel compiler, hence the final qualification enabling Eiffel tools to rely, for example, on compilation errors produced by a C compiler.

The "conventions of the underlying platforms" cited in condition 1 govern the rules on file names (in particular the interpretation of path delimiters such as / and \ on Unix and Windows) and, for an External_system_file name of the form *<some_file.h>*, the places in the file system where *some_file.h* is to be found.

External file semantics

An External_file_use in an external routine declaration specifies that foreign language tools, to process the routine (for example to compile its original code), require access to the listed files.

31.11 THE C INTERFACE SUBLANGUAGE

The first special sublanguage that we study, C_external, addresses the needs of applications developers who need sophisticated access to C mechanisms (also provided for C++). You can of course limit yourself to the mechanisms described so far, simply declaring an external routine as **external** "*C*". But to exert more control on how your Eiffel software uses C mechanisms, you may use a whole slate of special C interface facilities:

- You can specify that a certain external routine is implemented on the C side as a **macro**, saving the overhead of function calls.
- You can use an External_signature, as studied above, to force a certain type signature ("*prototype*") for the arguments and result of the C function in the Eiffel-generated C code.
- You can request specific **include files** for certain C functions, using the External_file_use construct just studied.
- You can directly access C structures ("structs") and their components.
- You can even include the C code of an external routine in line, removing the need to maintain two separate source files, an Eiffel class file and a C compilation unit (**.c** file).

The next paragraphs describe these possibilities. They are complemented by the C++-specific facilities of the following section.

Syntax specification

Here is the syntax specification for the C interface sublanguage. First we remind ourselves of the context:

← This appeared first on page 819.

External languages

External	≜	**external** External_language [External_name]
External_language	≜	Unregistered_language | Registered_language
Unregistered_language	≜	Manifest_string
External_name	≜	**alias** Manifest_string
Registered_language	≜	... | C_external | … Others …

Now the C_external case of Registered_language:

C externals

C_external	≜	’' **"** ' **C** ’[**inline**] [External_signature] [External_file_use] ' **"** '

The C_external mechanism makes it possible, from Eiffel, to use the mechanisms of C. The syntax covers two basic schemes:

- You may rely on an existing C function. You will not, in this case, use **inline**. If the C function's name is different from the lower name of the Eiffel routine, specify it in the **alias** (External_name) clause; otherwise you may just omit that clause.
- You may also write C code *within* the Eiffel routine, putting that code in the **alias** clause and specifying **inline**.

In the second case the C code can directly manipulate the routine's formal arguments and, through them, Eiffel objects. The primary application (rather than writing complex processing in C code in an Eiffel class, which would make little sense) is to provide access to existing C libraries without having to write and maintain any new C files even if some "glue code" is necessary, for example to perform type adaptations. Such code, which should remain short and simple, will be directly included and maintained in the Eiffel classes providing the interface to the legacy code.

The **alias** part is a Manifest_string of one of the two available forms:

- It may begin and end with a double quote **"**; then any double quote character appearing in it must be preceded by a percent sign, as **%"**; line separations are marked by the special code for "new line", **%N**.

← See "MANIFEST STRINGS", 29.8, page 784.

- If the text extends over more than one line, it is more convenient to use a Verbatim_string: a sequence of lines to be taken exactly as they are, preceded by **"[** at the end of a line and followed by **]"** at the beginning of a line.

In this Manifest_string, you may refer to any formal argument *a* of the external routine through the notation **$***a* (a dollar sign immediately followed by the name of the argument). For *a* you may use either upper or lower case, lower being the recommended style as usual.

We now explore these capabilities, and look further into how you can match Eiffel types with their C counterparts.

Specifying C code inline

In all the preceding mechanisms, the C code resides outside of the Eiffel text, in its own separate files. Although this separation of elements written in different languages is usually appropriate, you may not like the idea of having to look after different places, and find it easier to manage your software by keeping everything at the same place. It is indeed possible to include C code within the declaration of an external routine. This way you don't need to include any external C file in your system.

This possibility is appropriate mostly for short C routines concentrated in "wrapper" classes providing Eiffel interfaces to C libraries.

A C_special part may specify **inline**, optionally followed by the usual specifications of a C signature and include files. This indicates that the actual C text appears in the **alias** clause (External_name), which is required in this case. Here is an example including both an explicit signature and an include file (which might contain the declaration of a C variable *cvar*):

Warning: *the content of the* **alias** *clause represents C, not Eiffel.*

```
an_inline_function (x,y: INTEGER): INTEGER
        external "[
                C
                        inline
                        use <stdio.h>, /path/user/her_include.h
                ]"
        alias "[
                        if ($x > cvar) {
                                some_c_function ($y, cvar++);
                        }
                ]"
        end
```

The Manifest_string appearing in the **alias** clause is C code meant to be passed on exactly as it is (except for the replacement of elements in quotes, as explained next) to a C compiler. The most convenient way to express it is to use, as here, a Verbatim_string, so that all the lines between the initial **"[** and the final **]"** are plain C text, with no need for special codes to represent characters such as quotes, or to mark the beginning and end of a line.

The only exception to the verbatim interpretation of the string as C code is the convention allowing the C code to access entities from the enclosing Eiffel text. Any occurrence in the **alias** part of a substring of the form *$eiffel_entity*, where *eiffel_entity* is a formal argument of the routine or an attribute of the enclosing class, denotes the corresponding Eiffel entity, which the Eiffel compiler will replace by the appropriate access code for the benefit of the C compiler. *$x* and *$y* in the above extract are examples of this facility; they denote the function's *x* and *y* arguments.

This use of the $ operator is consistent with the Address form of arguments, serving to pass Eiffel features to external languages.

← *"PASSING THE ADDRESS OF AN EIFFEL FEATURE", 31.8, page 823*.

Note that *eiffel_entity* must follow the **$** sign with no intervening space. Any occurrence in the C text of a **$** sign not immediately followed by an Eiffel entity is considered C text to be taken verbatim.

Here is the validity rule for inline C functions:

C external rule *VZCC*

A C_external for the declaration of an external routine *r* is valid if and only if it satisfies the following conditions:

1 • At least one of the optional **inline** and External_signature components is present.

2 • If the **inline** part is present, the external routine includes an External_name component, of the form **alias** *C_text*.

3 • If case 2 applies, then for any occurrence in *C_text* of an Identifier *a* immediately preceded by a dollar sign **$** the lower name of *a* is the lower name of a formal argument of *r*.

C Inline semantics

In an external routine *er* of the **inline** form, an External_name of the form **alias** *C_text* denotes the algorithm defined, according to the semantics of the C language, by a C function that has:

- As its signature, the signature specified by *er*.
- As its body, *C_text* after replacement of every occurrence of **$***a*, where the lower name of *a* is the lower name of one of the formal arguments of *er*, by *a*.

Controlling the Eiffel-C type correspondence

In passing arguments to C functions, and getting results back into Eiffel entities, you need to know exactly how the types will match. Eiffel provides (through the C library of the supporting environments) a set of predefined C types used, by default, to represent the types of Eiffel values passed to and from external C routines. If you are writing external C functions specifically for use in connection with Eiffel software, you should use these types (obtained from a standard include file provided with the Eiffel delivery) to declare the functions’ arguments and results.:

Eiffel to C default type correspondence

Eiffel type	Corresponding C type with declaration	
BOOLEAN	typedef unsigned char	EIF_BOOLEAN
CHARACTER	typedef unsigned char	EIF_CHARACTER
INTEGER_8	typedef unsigned char	EIF_INTEGER_8
INTEGER_16	(16-bit integer)	EIF_INTEGER_16
INTEGER	(32-bit integer)	EIF_INTEGER_32

INTEGER_64	(64-bit integer)	EIF_INTEGER_64
REAL_32	(32-bit float)	EIF_REAL_32
REAL	(64-bit integer)	EIF_REAL
POINTER	typedef char ∗	EIF_POINTER
Any reference type	typedef char ∗	EIF_REFERENCE

The C type definitions given in parentheses are platform-dependent. For example "32-bit integer" will be typedef long on many platforms, but not all.

This will not work, however, if you are using pre-existing C functions, written without knowledge of Eiffel. In such a case the declarations will not match those generated by the Eiffel compiler using the correspondence above, and you may get C compilation errors. Fortunately, the type checking of C is more bark than bite. You can easily pacify it by "casting" the type of arguments and results, that is to say, specifying explicit types.

It would be unpleasant to have to do the casting manually on the C code (if only because we are, as noted, trying through all the facilities described here to limit the amount of C programming to be done). The External_signature facility is here to help. It allows you to specify the exact set of casting types for the arguments and result, so that the C compiler will find what it expects. Here is a typical use:

```
your_external (a, b: INTEGER): INTEGER
        external
                "C (int, int): EIF_INTEGER_32"
        end
```

This example assumes that the C function requires arguments of the C type *int* (integer) and returns a result also of that type, which must be cast into an *EIF_INTEGER_32*.

31.12 THE C++ INTERFACE SUBLANGUAGE

In addition to the mechanisms available to all external routines, all the C-specific techniques of the previous sections are available for use with C++ code. So is the Cecil library described in a later section and allowing external software to call Eiffel. In addition, the C++ interface sublanguage offers a number of specific mechanisms:

- You can create instances of C++ classes from Eiffel, using the C++ "constructor" of your choice.
- You can apply to these objects all the corresponding operations from the C++ class: executing functions ("methods"), accessing data members, executing destructors.

- You can use the **Legacy++** tool to produce an Eiffel "wrapper class" encapsulating all the features of a C++ class, so that the result will look to the rest of the Eiffel software as if it had been written in Eiffel.

The syntax specification

The C++-specific mechanisms come under the construct C++_external, one of the variants of Registered_language, itself one of the possibilities for External_language.

C++ externals

C++_external ≜ ' " ' **C++**
inline
[External_signature]
[External_file_use]
' " '

As in the C case, you may directly write C++ code which can access the external routine's argument and hence Eiffel objects. Such code can, among other operations, create and delete C++ objects using C++ constructors and destructors.

Unlike in the C case, this inline facility is the *only* possibility: you cannot rely on an existing function. The reason is that C++ functions — if not "static" — require a target object, like Eiffel routines. By directly writing appropriate inline C++ code, you will take care of providing the target object whenever required.

Conditions on C++ features

C++ external rule *VZC+*

A C++_external part for the declaration of an external routine *r* is valid if and only if it satisfies the following conditions:

1 • The external routine includes an External_name component, of the form **alias** *C++_text*.

2 • For any occurrence in *C++_text* of an Identifier *a* immediately preceded by a dollar sign **$**, the lower name of *a* is the lower name of a formal argument of *r*.

Processing C++ features

A C++_external, if present, indicates one of the following, all illustrated by examples in the next sections:

- If the special feature's declaration starts **function**, it indicates that the Eiffel feature will call a C++ *member function* (also known as a "method") from the class listed. The function's name is by default the same as the name of the Eiffel feature; as usual, you can specify a different name through the **alias** clause of the external declaration.
- If the declaration starts with **static**, it indicates a call to a C++ *static function*.
- If the declaration starts with **new**, it indicates a call to one of the *constructors* in the C++ class, which will create a new instance of that class and apply to it the corresponding constructor function.
- If the declaration starts with **delete**, it indicates a call to a *destructor* from the C++ class. In this case the Eiffel class will inherit from *MEMORY* and redefine the *dispose* procedure to execute the destructor operations whenever the Eiffel objects are garbage-collected.
- If the declaration starts with **data_member**, it indicates access to a *data member* (attribute in Eiffel terminology) from the C++ class.
- If it starts with **structure**, it provides the same facilities as C_structure.

The techniques for specifying signatures, external files and type correspondence are the same as for C.

← *"Specifying an external routine signature", page 828*; *"Specifying external files", page 830*; *"Controlling the Eiffel-C type correspondence", page 836*.

C++ Inline semantics

In an external routine *er* of the C++_external form, an External_name of the form **alias** *C++_text* denotes the algorithm defined, according to the semantics of the C++ language, by a C++ function that has:

- As its signature, the signature specified by *er*.
- As its body, *C++_text* after replacement of every occurrence of **$***a*, where the lower name of *a* is the lower name of one of the formal arguments of *er*, by *a*.

Extra argument

For a non-static C++ member function or destructor, the corresponding Eiffel feature should include an extra argument of type *POINTER*, at the first position. This argument represents the C++ object to which the function will be applied.

For example, a C++ function

```
void add (int new_int);
```

should have the Eiffel counterpart

```
cpp_add (obj: POINTER; new_int: INTEGER)
            -- Encapsulation of member function add.
      external "[
            "C++
                  member IntArray
                  signature (IntArray *, int)
                  use intarray.h
            ]"
      end
```

This scheme, however, is often inconvenient because it forces the Eiffel side to work on objects in a non-object-oriented way. (The O-O way treats the current object, within a class, as implicit.) A better approach, used by Legacy++, is to make a feature such as *cpp_add* secret, and to export a feature whose signature corresponds to that of the original C++ function, with no extra object argument; that feature will use a secret attribute *object_ptr* to access the object. In the example this will give

```
add (new_int: INTEGER)
            -- Encapsulation of member function add.
      do
            cpp_add (object_ptr, new_int)
      end
```

where *object_ptr* is a secret attribute of type *POINTER*, initialized by the creation procedures of the class. To the Eiffel developer, *add* looks like a normal object-oriented feature, which takes only the expected argument. Further examples appear below.

There is no need for an extra argument in the case of static member functions, constructors and data members.

The next section will illustrate the various available possibilities by showing the code generated, in each case, by the Legacy++ tool.

31.13 WRAPPING C++ CLASSES: LEGACY++

Legacy++ is a tool, not a part of the language specification. Its practical role is, however, sufficiently important to justify a special section in this chapter. This will also provide us with a set of examples covering all the special C++ encapsulation possibilities.

The role of Legacy++

Often you will want to provide an Eiffel encapsulation of **all** the facilities — member functions, static functions, constructors, destructors, data members — of a C++ class. This means producing an Eiffel class that will provide an Eiffel feature for each one of these C++ facilities, using external declarations based on the mechanisms listed in the preceding section.

Rather than writing these external declarations and the class structure manually, you can use Legacy++ to produce the Eiffel class automatically from the C++ class.

Calling Legacy++

Legacy++ is called with an argument denoting a **.h** file that must contain C++ code: one or more classes and structure declarations. It will translate these declarations into Eiffel wrapper classes.

The following options are available:

- **–E**: apply the C preprocessor to the file, so that it will process #include, #define, #ifdef and other preprocessor directives. This is the default.
- **–NE**: do not apply the C preprocessor to the file.
- **–p** *directories*: use *directories* as include path.
- -**–c** *compiler*: use *compiler* as the C++ compiler.
- **–g**: treat the C++ code as being intended for the GNU C++ compiler.

Result of applying Legacy++

Running Legacy++ on a C++ file will produce the corresponding Eiffel classes. Legacy++ processes not only C++ classes but also C++ “structs”; in both cases it will generate an Eiffel class. Among its properties:

- Legacy++ knows about *default specifiers*: **public** for classes, **private** for structs.
- Legacy++; will generate Eiffel features for *member functions* (static or not).

- It will also handle any *constructors* and *destructors* given in the C++ code, yielding the corresponding Eiffel creation procedures. If there is no constructor, it will produce a creation procedure with no arguments and an empty body.

- For any non-static member function or destructor, Legacy++ will generate a *secret feature* with an extra argument representing the object, as explained in the preceding section. It will also produce a public feature with the same number of arguments as the C++ function, relying on a call to the secret feature, as illustrated for *add* and *cpp_add* above.

- The *char* ∗ type is translated into *STRING*. Pointer types, as well as reference types corresponding to classes and types that Legacy++ has processed, will be translated into *POINTER*. Other types will yield the type *UNRESOLVED_TYPE*.

Legacy++ limitations

It is up to you to supply Eiffel equivalents of all the needed types. If Legacy++ encounters the name of a C++ class or type that is does not know — it is neither a predefined type nor a previously translated class — it will use the Eiffel type name *UNRESOLVED_TYPE*. If you do not change that type in the generated class, the Eiffel compiler will report an error.

Legacy++ does not handle inline function declarations and makes no effort to understand the C++ inheritance structure. More generally, given the differences in the semantic models of C++ and Eiffel, Legacy++ can only perform the basic Eiffel wrapping of a C++ class, rather than a full translation. You should always inspect the result and be prepared to adapt it manually. Legacy++'s contribution is to take care of the bulk of the work, in particular the tedious and repetitive parts. The final details are left to the Eiffel software developer.

Legacy++ example

Consider the following C++ class, which has an example of every kind of facility that one may wish to access from the Eiffel side:

Warning: this is C++, not Eiffel.

```
class IntArray
    {
    public:
        IntArray (int size);
        ~IntArray ();
        void output ();
        void add (int new_int);
        static char * type ();
    protected:
        int *_integers;
    };
```

Here is the result of applying Legacy++ to that class, which will serve as an illustration of both the C++ interface mechanisms and Legacy++:

```
note
    description:
     "Eiffel encapsulation of C++ class IntArray"
class
    INTARRAY
inherit
    MEMORY
     redefine
        dispose
     end

create
    make
feature -- Initialization
    make (size: INTEGER)
        -- Create Eiffel and C++ objects.
     do
        object_ptr := cpp_new (size)
     end

feature -- Removal
    dispose
        -- Delete C++ object.
     do
        cpp_delete (object_ptr)
     end
```

```
feature
	output
			-- Call C++ counterpart.
		do
			cpp_output (object_ptr)
		end

	add (new_int: INTEGER)
			-- Call C++ counterpart.
		do
			cpp_add (object_ptr, new_int)
		end

feature {INTARRAY}
	underscore_integers: POINTER
			-- Value of corresponding C++ data member.
		do
			Result := underscore_integers (object_ptr)
		end

feature {NONE} -- Externals
	cpp_new (size: INTEGER): POINTER is
			-- Call single constructor of C++ class.
		external"[
				C++ new IntArray
				signature (EIF_INTEGER_32) use INTARRAY.h
			]"
		end

	cpp_delete (cpp_obj: POINTER)
			-- Call C++ destructor on C++ object.
		external"[
				C++ delete IntArray
				 signature () use INTARRAY.h
				"]
		end

	cpp_output (cpp_obj: POINTER)
			-- Call C++ member function.
		external "[
				C++ function IntArray
				signature () use INTARRAY.h
				]"
		alias
			"output"
		end
```

```
    cpp_add (cpp_obj: POINTER; new_int: INTEGER)
        -- Call C++ member function.
     external "[
          C++ function IntArray
           signature (EIF_INTEGER_32) use INTARRAY.h
          ]"
     alias
        "add"
     end

    cpp_underscore_integers (cpp_obj: POINTER): POINTER
        -- Value of C++ data member
     external "[
          C++ data IntArray
           use INTARRAY.h
          ]"
     alias
        "integers"
     end

feature {NONE} -- Implementation
    object_ptr: POINTER
        -- Access to C++ object
end
```

31.14 USING DYNAMIC LINKE LIBRARIES (DLLS)

Dynamic Link Libraries enable an Eiffel system to take advantage of DLL routines on platforms (such as Windows) supporting the DLL mechanism. A DLL routine is not compiled into your system but kept separate; your system will load the routine the first time it needs to call it. This has two principal advantages:

- You pay only, in memory usage, for what you use. Without DLLs every system must be compiled with every piece of functionality it *ight* use even if 98% of executions don't need it. This is a source of size bloat.
- DLLs facilitate software evolution since you can deliver incremental functionality updates through specific DLL replacements, without chaning the entire system previously delivered to your users.

Each of these advantages also implies less pleasant counterparts (leading to the phrase "*DLL hell*"): unlike with statically linked systems, a missing component may not be detected until run time (and in certain executions only); a product may install a new DLL that invalidates another product; and you never quite know what your users' configuration is, which doesn't facilitate customer support. DLLs are, however, a very popular technique. ISE Eiffel includes a DLL tool for generating DLLs from Eiffel systems.

Eiffel systems also need to *use* DLLs produced elsewhere. Two mechanisms are available for that purpose:

- A DLL sublanguage, similar in spirit to the C and C++ sublanguages reviewed previously, lets you specify DLL routines that you need. Although based on dynamic linking this is a "static" mechanism in that you have to express what you need in your software, before compiling.
- There is also a completely dynamic mechanism, DESC, allowing you to wait until run time to determine what dynamic libraries you need and what routines you want to call.

We now review these two mechanisms in turn.

The static DLL sublanguage

Using the DLL sublangage you can define an external Eiffel routine relying on a routine from a DLL. You will use a clause **external dll** *file_name* to specify the *file_name* for the dynamic library, and a clause **alias** *name* to specify the name or integer index of the desired routine in that library.

Here is an Eiffel routine encapsulating a function from a DLL:

```
dynamic_external (a, b, c: INTEGER)
        external "[
                "dll
                        signature (WORD, DWORD, WORD)
                        use herlib.dll
                ]"
        alias
                "35"
        end
```

A **dll** subclause requires you to specify a **DLL index or name**, indicating where to find the routine in the DLL. Use the **alias** part for that purpose. Normally, as we have seen, the **alias** part of an External declaration gives the native name of the routine (required only if different from the Eiffel name). In the case of a DLL it is also acceptable to provide the routine's index in the library, an integer, such as *35* in the example. There is no ambiguity: an integer alias denotes an index, anything else is taken as a name. This variant also requires the presence of an External_signature part.

The **alias** *part also gives the C text of an* **inline** *routine..*

Iif your system uses several routines from the same DLL, its execution will only load one instance of the DLL. When the execution terminates, the Eiffel run-time system will free all DLL instances loaded in this way.

Here is the syntax for the DLL variant of the **external** part:

DLL externals

```
DLL_external ≜ ' " ' dll
               [windows]
               DLL_identifier
               [Blanks_or_tabs DLL_index]
               [External_signature]
               [External_file_use]
               ' " '

DLL_identifier ≜ Simple_string

DLL_index ≜ Integer
```

Through a DLL_external you may define an Eiffel routine whose execution calls an external mechanism from a Dynamic Link Library, not loaded until first use.

The mechanism assumes a dynamic loading facility, such as exist on modern platforms; it is specified to work with any such platform.

External DLL rule *VZDL*

A DLL_external of DLL_identifier *i* is valid if and only if it satisfies the following conditions:

1 • When interpreted as a file name according to the conventions of the underlying platform, *i* denotes a file.

2 • The file is accessible for reading.

3 • The file's content denotes a dynamically loadable module.

External DLL semantics

The routine to be executed (after loading if necessary) in a call to a DLL_external is the dynamically loadable routine from the file specified by the DLL_identifier and, within that file, by its name and the DLL_index if present.

The DLL mechanism specified here is **static** since it requires you to indicate, in the software text, the name of the library and the index (in the form of an integer constant) of the desired routine in that library. One of the advantages of DLLs is the ability to wait until run time to specify both the library and the routine. A corresponding **dynamic** mechanism, complementing the facilities just described, is also available through the DESC library studied later in this chapter.

→ *"DESC: CALLING A DLL ROUTINE DETERMINED AT RUN TIME", 31.15, page 848.*

The optional **windows** qualifier specifies that the DLL uses the calling conventions of the Windows platform.

31.15 DESC: CALLING A DLL ROUTINE DETERMINED AT RUN TIME

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), or the routine itself in the C **inline** case. Even the C **dll** mechanism requires you to specify the name of the Dynamic Link Library and the index of the desired routine.

The **Dynamic External Shared Call** mechanism (DESC for short) removes this limitation by letting you 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.

DESC is a library, not a language mechanism, but as important in practice as the purely linguistic mechanisms defined in this chapter.

In line with the general spirit of Eiffel, the DESC takes care of low-level aspects of DLL programming, relieving 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.

DLLs vary with operating systems. The description in this section applies to Windows.

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* and *DLL_ROUTINE*, supported by an auxiliary class *SHARED_LIBRARY_CONSTANTS*, provide the basis of DESC:

- An instance of class *DLL* describes a 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_ROUTINE* describes a routine from a DLL. The class has an attribute of type *DLL* 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 some 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.

The normal sequence of operations to use the DESC mechanism is:

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

2 • Create a routine object (an instance of *DLL_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.

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

You may repeat each of these steps as often as necessary to use multiple libraries, multiple routines in a library, or multiple calls to a given routine. More details follow.

Creating a library object

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

```
your_dll: DLL
```

replacing *your_dll* by whatever name you have chosen to denote the library in your software; execute a creation instruction of the form

```
create 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*, the value of *meaningful* is true if and only if *error_code* = 0.

Creating a routine object

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

```
your_routine: DLL_ROUTINE
```

replacing *your_routine* by the name you have chosen to denote the routine in your software, and execute a creation instruction of the form

```
create 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:

```
create your_routine.make_by_index
            (your_dll,
            your_routine_index,   -- The only differing argument
            [argtyp1, argtyp2, ...],
            res_type)
```

In either form *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, 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_ROUTINE*, the value of *meaningful* is true only if *error_code* = 0.

Procedures *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*, with *make_by_name*; by its index, of type *INTEGER*, with *make_by_index*.
- 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. Possible type codes appear next.
- 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 through the notation [*a1*, *a2*, ...] ; here the array items *argtyp1*, *argtyp2*, ... must all be integers giving the type codes of the successive arguments to the routine, taken from the list appearing next. (Use an empty manifest array, [ ], if the routine has no arguments.)

Type codes

For the type codes used in the array serving as third argument to *make_by_name* and *make_by_index*, and in the fourth argument *res_type* , the class *SHARED_LIBRARY_CONSTANTS* provides a set of constant integer attributes; the easiest way to let a class use them is to make it an heir of that library class. Here is the list of codes:

Type code	Meaning and comments
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 value**. Passed to C as unsigned character: 0 for false, nonzero for true.
T_character.	**Character value**.
T_integer	**Long integer**.
T_no_type	**No type**. Useful for *res_type* in the case of a procedure (which has no result type).
T_real	**Real number**.
T_pointer	**Pointer** to C structure.
T_reference	**Reference** to Eiffel object.
T_short_intege r	**Short integer**. The Eiffel side will use normal *INTEGER* values for the corresponding actual arguments.
T_string	**String**. What is passed to C is the C form of the Eiffel string, obtained through the feature *to_c* of class *STRING*. To pass the Eiffel string object, use *T_reference*.

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_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])
```

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_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.integer_result	*INTEGER*
your_routine.integer_result	*INTEGER*
your_routine.real_result	*REAL*
your_routine.reference_result (To use the result, an assignment attempt will usually be necessary.)	*ANY*
your_routine.string_result (Result converted to Eiffel string format through the feature *from_c* of class *STRING*.)	*STRING*

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 *ANY*. The third precondition clause of procedure *call* states it:

```
call (args: ARRAY [ANY])
        require
                meaningful: meaningful
                valid_array: args /= Void
                conformant: conforms_to_signature (args)
```

This precondition, combined with queries *meaningful* and *error_code* in classes *DLL* and *DLL_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.

Sharing and freeing

One of the effects of creating a library object through a creation instruction of the form **create** *your_dll*.*make* ("*your_lib_name*") is, as noted, 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.

You may, if you wish, load several instances of a given 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*, two different instances will be loaded.

When a DESC library object is no longer accessible and the garbage collector reclaims it, this will automatically (through the procedure *dispose* of class *MEMORY* as redefined for class *DLL*) 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_dll*.*meaningful* for any routine object *your_routine* that was created relative to *your_dll*.

31.16 THE CECIL LIBRARY

The mechanisms studied so far support **call-out**: calling foreign mechanisms from Eiffel. There is a complementary need for a **call-in** mechanism, enabling foreign software to call Eiffel features.

Cecil overview

Call-in and call-out are in fact closely related since an external (call-out) routine may pass, among others, arguments of the Address form, denoting features of the enclosing class. The sole purpose of such arguments is, obviously, to let foreign routines call the associated Eiffel features.

← *"PASSING THE ADDRESS OF AN EIFFEL FEATURE", 31.8, page 823.*

More generally, some developers may wish to write foreign routines that create Eiffel objects and apply features to these objects, without necessarily 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**. The first C in the acronym is there mostly for historical reasons: you can use Cecil from any foreign language that supports standard argument passing conventions.

Cecil role and status

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 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. The Cecil library this section describes, then, is not part of Eiffel as a language, but it is a required component of any Eiffel implementation.

The following Cecil resources should complement the explanations of this section:

• http://eiffel.com/doc/manuals/library/cecil/ is a complete Cecil manual.

- If you program non-trivial Cecil applications you will benefit from the set of examples at ftp://ftp.eiffel.com/pub/examples/cecil; you can retrieve individual examples from that directory, or download all examples, zipped, from ftp://ftp.eiffel.com/pub/examples/cecil/cecil.zip. The directory is split into two subdirectories: *unix-examples* and *windows-examples*.

Compiling for Cecil

To use the facilities of an Eiffel system through Cecil you must first compile a "cecilized" form of it. This may require a special compilation or (as with ISE Eiffel) you may simply get the "cecilized" form as a standard output of your compilation with no extra work.

You will of course need to compile your foreign application, a process that is not always as automatic as Eiffel compilation as managed by good Eiffel environments. Even here, however, Eiffel can help: you can specify a Make file in the **external** part of your Ace through a directive of the form

```
external: make: "your_makefile"
```

which causes Eiffel compilation to start C compilation using the provided Make file. (To specify its location, remember that you can use environment variables, such as $EIFFEL5 denoting the location of the Eiffel installation, in the Ace file.)

As explained next, the foreign software will gain access to the Cecil mechanisms through two include files produced by the Eiffel environment: eif_cecil.h and (if execution starts on the foreign side rather than from Eiffel) eif_setup.h. You will use the "include" option of your C compiler, normally –I, to specify the directory where these files reside.

→ For the location of this directory in ISE Eiffel see "ISE Eiffel specifics", page 866.

Avoiding abusive optimization

Even with a compiler that generates cecilized code without any special compilation option, you may have to exert some care if the compiler (again such as ISE Eiffel) performs dead-code-removal optimization, to delete the generated code for routines that are not called from within the Eiffel system. Such routines may still be needed by foreign software as part of the cecilized interface. To protect them from over-enthusiastic dead code removal, list them in the **visible** clause of the Ace file, as in

```
system system_name root ... default ... cluster
        ...
        your_cluster: "/home/user/cluster1"
                adapt
                        ...
                visible
                        CLASS1
                        CLASS2
                                create
                                        "other_make"
                                export
                                        "feat1", "feat2"
                                end
                end
        ... Other cluster specifications...
end
```

→ *See appendix B about Lace, in particular "VISIBLE FEATURES", B.13, page 1024*

Here all exported features of *CLASS1* are available to the external software; for *CLASS2*, only *other_make* (for creation) and *feat1* and *feat2* (for normal call) are available.

By default the status of features is deduced from the Eiffel class text: only the publicly available features will be available through the Cecil interface. You can use the **export** clause to override this default, in particular to make a feature is available to the outside world even though it is not used in the Eiffel system and hence subject to dead-code removal.

The creation status is determined in a similar way: by default any procedure listed in Eiffel as a generally available for creation will be accessible through Cecil; you can override this default through the **create** subclause of the **visible** clause.

Note that because a Cecil application will create and initialize an object through two separate calls (unlike the Eiffel instruction *a*.*make* (…) which does boty), the creation and export status are the same for Cecil, so listing a feature under **create** or **export** has the same effect: making it available to foreign software through the Cecil interface.

Basic Cecil conventions

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_PROCEDURE**, explained below. Their declarations appear in a C "header file", *eif_cecil*.*h*, which you may add to a C program through the C preprocessor directive

Eiffel's emphasis on clarity suggests using **eiffel_** *and* **EIFFEL_** *as prefixes, but some of the resulting names would be too long for some C compilers.*

```
#include "eif_cecil.h"
```

Warning: this is C, not Eiffel.

A similar mechanism will be available for other supported foreign languages, although the rest of the discussion will assume C or C++.

We now review the various facilities available from *cecil*.*h*. To avoid any confusion with the format used in the rest of this book for Eiffel software elements, C code will appear as follows (in color):

- Bold font (as elsewhere for Eiffel keywords) for Cecil functions, macros and types, such as **eif_type_id** and **EIF_PROCEDURE**.
- Italic font, for C names representing Eiffel class names or entities, such as *CLASS_NAME*.
- Regular font for ordinary C text, including example variables illustrating function usage, such as your_id.

The basic scheme of using Cecil is the following:

- Build an Eiffel system.
- "Cecilize" it: compile it for Cecil use. This may require some specific compilation options, or at least, as noted above, protecting features from dead code removal.
- Write a program in C or some other language that gains access to the resulting facilities through appropriate **include** directives and uses Cecil functions and macros to create Eiffel objects, call features on them, and receive any resulting exceptions.

Initializing the Eiffel 4 run-time

An application using Cecil, involving both Eiffel and foreign elements, may start its execution from either side. If execution starts on the non-Eiffel side — in other words, if the foreign language is in control — it will need, prior to calling any Eiffel facility, to set up the Eiffel run time to ensure that Eiffel mechanisms such as garbage collection and signal handling will work properly. It will also need, before it terminates, to call the run-time termination mechanisms, ensuring in particular that all Eiffel objects are freed and the corresponding *dispose* procedures are called to free any associated system resources.

The runtime setup will typically appear in the foreign application's main program. Simply add the preprocessor directive

```
#include "eif_setup.h"
```

Warning: this is C, not Eiffel.

To start the Eiffel runtime, use

```
EIF_INITIALIZE (failure_function);
```

Warning: this is C, not Eiffel.

where *failure_function* () is a function to be called in case of failure to initialize. To terminate the Eiffel runtime, collect all objects and call their *dispose* procedures if any, use

```
EIF_DISPOSE_ALL;
```

Warning: this is C, not Eiffel.

EIF_INITIALIZE and EIF_DISPOSE_ALL are macros defined in eif_setup.h. The macros assume that the enclosing function, normally the main program, has the three standard arguments, as in

```
main (int argc, char **argv, char **envp);
```

Warning: this is C, not Eiffel.

Manipulating values of basic Eiffel types

If you pass Eiffel values of basic types (integers, booleans and so on) you will need to make sure that the C side manipulates them properly. For example there is no guarantee that an Eiffel *INTEGER* and a C int are the same; for portability and to guarantee numerical precisions the Eiffel-C interface includes the following set of macros defining the C representation of the Eiffel basic types:

EIF_BOOLEAN	**EIF_CHARACTER**	**EIF_INTEGER_8**
EIF_INTEGER_16	**EIF_INTEGER_32**	**EIF_INTEGER_64**
EIF_REAL_32	**EIF_REAL**	**EIF_POINTER**

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

The macro **EIFFEL_TYPE** denotes the C type (actually int) covering C representations of Eiffel types; the possible values are the twelve listed, plus **EIF_REFERENCE**, introduced below.

If you have control over the C code, always use the above types to manipulate Eiffel values from C. So with an Eiffel external function

```
c_func (ptr: POINTER; obj: OBJECT): INTEGER is
        external
                "C include %"your_file.h%""
        end
```

you may write the C side as

```
EIF_INTEGER_32 c_func (EIF_POINTER ptr, EIF_OBJECT obj)
        {… Function body …}
```

Warning: this is C, not Eiffel.

In other cases, the C function pre-exists and you cannot (or do not want to) change it. In that case you should take care of the proper typing on the Eiffel side, using the External_signature facility introduced earlier in this chapter With a function

←See *"Controlling the Eiffel-C type correspondence", page 836*.

```
int other_func (void *arg1, char c, FILE *file)
        {… Function body …}
```

Warning: this is C, not Eiffel.

you should write the Eiffel external as

```
other_func (arg1: POINTER; c: CHARACTER; file: POINTER):
                INTEGER
        external
                "C (void *, char, FILE *) : int include %""your_file.h%""
        end
```

Omitting the External_signature part (the part that lists the C types before the colons) would produce C compilation warnings and possibly errors.

Manipulating Eiffel class types

To call Eiffel features, the foreign software will need to access the classes and types to which they belong. It will know an Eiffel type through a "type-id", of type **EIF_TYPE_ID**.

To obtain a type-id for a type *TYPENAME* and record it in a C variable your_id, use the function **eif_type_id**, returning an **EIF_TYPE_ID**:

Warning: this is C, not Eiffel.

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

← "Avoiding abusive optimization", page 856.

As usual, you must make sure that the base class of *TYPENAME* is not optimized away by the compiler.

If the class is generic, include the generic parameters in the *TYPENAME* as in:

Warning: this is C, not Eiffel.

```
your_other_id = eif_type_id ("ARRAY [INTEGER]");
```

Given an Eiffel type descriptor type_id of **EIF_TYPE_ID**, you can obtain the corresponding Eiffel type name as well as the name of the generating class (the type's base class). Use **eif_type** (tid) for the type name and **eif_class** (tid) for the class name. In both cases the result is a char ∗, representing a C string.

Accessing an Eiffel object

A foreign 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_OBJECT**.

Current C guidelines suggests that **eif_access** *should return a* void ∗

A value your_object of type **EIF_OBJECT** is not 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_OBJECT** and returns a pointer to the object:

Warning: this is C, not Eiffel. The result is a null pointer if your_object *represents 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_OBJECT**, **eif_access** will retrieve a correct pointer. If the implementation does not move objects, **eif_access** will do little or no work.

The result type of **eif_access** is of type **EIF_REFERENCE**. A value of this type is a pointer to an Eiffel object; you can pass it to an Eiffel routine, or as the result of a C external. Do not, however, pass an **EIF_REFERENCE** to another C function, since the object might have moved; use **EIF_OBJECT** instead.

What if your_object is a variable that 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:

```
EIF_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);
```

Warning: this is C, not Eiffel.

A call to **eif_wean** actually returns a value: an **EIF_REFERENCE** to the object just “weaned”.

You should use **eif_adopt** for a value of type **EIF_OBJECT**, created by an Eiffel routine and passed as argument to the foreign software. For an **EIF_REFERENCE** value returned by one of the Cecil mechanisms, use **eif_protect** instead. An example appears next with an **EIF_REFERENCE** denoting an Eiffel string created by **eif_string** ("*SOME TEXT*"). Function **eif_protect** returns an **EIF_OBJECT**; as with **eif_adopt**, you should **eif_wean** that **EIF_OBJECT** when you do not need it any more.

Creating an Eiffel object

To create an object from outside, use the function **eif_create**, which takes an **EIF_TYPE_ID** argument and returns an **EIF_OBJECT**. For example:

Warning: this is C, not Eiffel.

```
EIF_OBJECT your_array;
...
your_array = eif_create (eif_type_id ("ARRAY [INTEGER]"));
```

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.

← About creation rules in Eiffel and the Creators clause, see chapter 20.

As the example shows, **eif_create** **does not call a <u>creation procedure</u>**. To apply a creation procedure, you will need to include a separate call, using function **eif_procedure** 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 cannot occur in Eiffel).

A shortcut is available for the case of string objects. As you will recall, *STRING* is a normal class with its own creation procedures. To avoid going through the creation of a *STRING* object and separate initialization, you can use **eif_string** as in:

```
 EIF_REFERENCE your_string;
EIF_OBJECT your_string_object;
my_string =  eif_string ("SOME TEXT");
your_string_object = eif_protect ("my_string");
```

The result of **eif_string** is an **EIF_REFERENCE**; if you are going to use it beyond the immediate context, make sure to **eif_protect** it as shown. When you do not need it any more, call **eif_wean** (your_string_object) to let the Eiffel garbage collector reclaim it once the Eiffel side is also done with it.

As a related facility, you can produce an Eiffel array eif_array from a C array c_array through the macro call

```
eif_array_from_c (eif_array, c_array, n, type_id)
```

where n, an integer, is the number of array elements and type_id, an integer, represents is the type of the array elements. The argument eif_array must be an **EIF_REFERENCE** denoting an array; c_array must be of type (type_id ∗), with enough space available to hold the array values. The value of type_id must be one of the Eiffel-C interface types defined earlier: **EIF_BOOLEAN** etc. for basic types, **EIF_REFERENCE** for any reference type.

← *"Manipulating values of basic Eiffel types", page 860; "Manipulating Eiffel class types", page 861.*

You can similarly use **eif_string_from_c** (eif_string, c_string, n) to get the C string (char ∗) equivalent of an Eiffel string.

Calling routines

Having gained access to Eiffel objects, the foreign application will want to apply Eiffel routines and attributes to them. To do so it needs 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_array as shown above, use the following to assign to variable your_procname a pointer to the Eiffel procedure whose Eiffel name in class *ARRAY* is *put*:

Warning: this is C, not Eiffel.

```
EIF_PROCEDURE your_array_put:
...
your_array_put = eif_procedure ("put", your_array);
```

Function **eif_procedure** is one of a group of functions, each corresponding to a different category of Eiffel routines: procedures, functions returning results of basic types, class types, bit types. Here is the list of these functions, with their argument and result types:

All these routines have the same arguments: a string (char ∗ in C), representing a routine name, and a type-id, obtained through *eif_type_id*.

→ See "Requesting a non-existing routine", page 865 below about Warning: this is C, not Eiffel.

These functions look for a routine of name rout_name in the base class of the type corresponding to type_id. If such a routine exists, the result will be a pointer to a C function representing it desired routine; you may then call that function on appropriate arguments. For example:

```
EIF_PROCEDURE eif_procedure
        (char * rout_name, EIF_TYPE_ID type_id)
EIF_REFERENCE_FUNCTION eif_reference_function
        (char * rout_name, EIF_TYPE_ID type_id)
EIF_INTEGER_32_FUNCTION eif_integer_32_function
        (char * rout_name, EIF_TYPE_ID type_id)
EIF_CHARACTER_FUNCTION eif_character_function
        (char * rout_name, EIF_TYPE_ID type_id)
EIF_REAL_32_FUNCTION eif_real_function
        (char * rout_name, EIF_TYPE_ID type_id)
EIF_REAL_FUNCTION eif_real_function
        (char * rout_name, EIF_TYPE_ID type_id)
EIF_BIT_FUNCTION eif_bit_function
        (char * rout_name, EIF_TYPE_ID type_id)
EIF_BOOLEAN_FUNCTION eif_boolean_function
        (char * rout_name, EIF_TYPE_ID type_id)
EIF_POINTER_FUNCTION eif_pointer_function
        (char * rout_name, EIF_TYPE_ID type_id)
```

Warning: this is C, not Eiffel.

The word **POINTER** *in* **EIF_POINTER_FUNCTION** *refers to the Eiffel POINTER type (see 31.8 above), not to C pointers.*

Variants of **eif_integer_32_function** *also exist for 8, 16 and 64.*

```
(your_array_put) (eif_access (your_array), 365, 10)
```

This applies the routine corresponding to *go*, accessible through your_array_put as a result of the above call to **eif_procedure**, to the object corresponding to your_array, with the actual argument 10. The corresponding call would have been written in Eiffel as *your_array*.*put* (345, *10*). In C, do not forget to enclose the name of the function pointer, here your_array_put, in parentheses, and to use **eif_access**.

As in Eiffel, the call will use dynamic binding: it will trigger the version of the feature directly adapted to the type of the target object.

Requesting a non-existing routine

The facilities just reviewed — **eif_procedure**, **eif_reference_function** and so on — enable the foreign side to gain access to an Eiffel feature. What if the requested feature does not exist in the class specified? If you stay within Eiffel this case will not arise since the type checking mechanism will detect the error at compile time; but from a foreign language no such static check is possible; the error will only become manifest at run time.

For the outcome in such a case you have a choice between two behaviors, which you can enforce by calling either of two status-setting procedures (whose effect will last until a call to the other):

- You can ensure that a request for a non-existent feature will trigger an exception, passed as a signal to the foreign side. This is not the default behavior, but you can obtain it by calling **eif_enable_visible_exception**.
- By default, functions such as **eif_procedure** and consorts return a null value if they can't find the Eiffel feature. You can restore this default behavior by calling **eif_disable_visible_exception**.

Accessing field objects

The macro **eif_attribute** enables the foreign side to access fields of objects, corresponding to attributes of the generating classes.

You may use the result of **eif_attribute** in two different ways: as an expression, or "r-value" in C terminology; or as a Variable entity, or "l-value", which may then be the target of an assignment. Such an assignment will re-attach the corresponding object field.

The macro requires four arguments:

```
eif_attribute
    (EIF_REFERENCE object, char * attrib_name,
    EIFFEL_TYPE type_id, int const * status);
```

Warning: this is C, not Eiffel.

The object argument denotes the object of which you want to access a field.; attrib_name denotes the name of the attribute in the generating class.

The third argument, type_id, serves to cast the result to the appropriate type. It must be one of the Eiffel-C interface types defined earlier: **EIF_BOOLEAN** etc. for basic types, **EIF_REFERENCE** for any reference type. **EIFFEL_TYPE** covers all these type values. In **EIF_REFERENCE** case, do not forget to **eif_protect** it the result if you will use it further.

← See "Manipulating values of basic Eiffel types", page 860, which also introduced **EIFFEL_TYPE**, *and "Manipulating Eiffel class types", page 861.*

The last argument, status, is a result code. Possible values are *status = **EIF_CECIL_OK**, indicating success, **EIF_NO_ATTRIBUTE**, indicating that no field exists in the object for the given name, and **EIF_CECIL_ERROR** for other Cecil errors. If you have selected **eif_enable_visible_exception** as explained above, the last two cases will trigger an exception.

ISE Eiffel specifics

The following comments apply to the use of Cecil with ISE Eiffel and may not be relevant for other implementations.

To will gain access to the Cecil facilities through two include files, both in $EIFFEL5/bench/spec/$PLATFORM/include where $EIFFEL5 is the Eiffel installation directory and $PLATFORM the platform code (such as windows, linux etc.):

- To use Cecil in a C file it suffices to include eif_eiffel.h.
- The main program may include eif_setup.h to access facilities for setting up and terminating the Eiffel run-time. This is not necessary if execution starts on the Eiffel side; if, however, a C main program starts execution and needs at some stage to call Eiffel mechanisms it will need these facilities to get everything initialized on the Eiffel side.

The following Lace options will be useful on Windows:

- Use console_application (yes) if you want to produce a console application rather than a default (graphical) Windows application.
- Use C_main ("*path_name*") to specify that the main program will be the C file at *path_name*.

ISE Eiffel offers three compilation modes: melted (super-fast incremental recompilation, no C generation), frozen (incremental, C generation), finalized (full C generation, extensive global optimizations). You can use Cecil with all three modes.

In the case of a melted system of name *system_name*, you must copy the file <*system_name*.melted> from the subdirectory EIFGEN/W_code of your project directory to the directory from which you will execute your C program. (The execution directory, not the compilation directory). This file will change after each melting; so on Unix it may be more convenient to use instead a symbolic link to it, which also saves space.

A limitation exists in case of a melted system: it is not permitted to use through Cecil any routine that has been melted in the last compilation. This would raise the run-time exception "**$** *applied to melted routine*". The solution is simple: refreeze.

To "cecilize" your system you do not need to use any special Eiffel compilation option. The only extra concern you need to have is, in finalized mode, to protect features from the dead-code removal algorithm, as explained earlier. Compilation produces both C code and a Makefile, in a subdirectory of EIFGEN in your project directory: EIFGEN/W_code (in melted or frozen mode) or EIFGEN/F_code (in finalized mode). To produce a CECIL library, you must, in a DOS console (Windows) of shell (Unix), go to the appropriate EIFGEN/*x*_code directory and run the make utility with the *cecil* option: *make cecil* (Unix), *nmake cecil* (Windows with Visual C++ and compatible compilers).

This generates a Cecil archive whose name derived from the name *system_name* of your Eiffel system: *system_name*.*lib* (Windows), lib*system_name*.a (Unix). The archive will include the Eiffel runtime thanks to the **include** directives listed above. Then it suffices to link the archive with the rest of your application through the link command appropriate for your operating system.

On Unix, you should use the –lm option to the link command to include the C mathematical library, required by the Eiffel runtime. You may need other libraries too, for example –lbsd on Linux, –lpthread (Posix threads) on Linux, –lthread (Solaris thread library) on Solaris. The linking command might look like this:

```
ld –lm –lbsd your_application.c libsystem_name.a
```