F

The C++ interface and Legacy++

F.1 OVERVIEW

Eiffel software must interact with software written in other languages. The previous appendix described how to access C software from Eiffel; in the present one you will learn about how to incorporate C++ classes in your Eiffel software.

The C++ interface offers the following 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.

To profit best from this appendix, it will be useful to be familiar with the essentials of the C interface as described in the preceding appendix.

The discussion concentrates on using C++ software from Eiffel. In the other direction, as noted in the preceding chapter, you can use the *Cecil* library (C-Eiffel Call-In Library) described in chapter 24 of *Eiffel*: *The Language* with important complements at *ftp://ftp.eiffel.com*.

F.2 THE SYNTAX SPECIFICATION

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

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

External	≜ **external** Language_name [External_name]
Language_name	≜ ’"’ Basic_language_name [Special_external_declaration] [Signature] [Include_files] ’"’
Basic_language_name	≜ "C" | "C++"
Special_external_declaration	≜ "[" Special_feature "]"
Special_feature	≜ Special_C_feature | Special_C++_feature
Special_C_feature	≜ ... See appendix E...
Special_C++_feature	≜ Member_function_call| Static_function_call | Object_creation Object_deletion Data_member_access
Member_function_call	≜ C++_Class
C++_Class	≜ Class_name File_name
Class_name	≜ Identifier
Static_function_call	≜ **static** C++_Class
Object_creation	≜ **new** C++_Class
Object_deletion	≜ **delete** C++_Class
Data_member_access	≜ **data_member** C++_Class
	-- The remaining elements are repeated -- from the C interface specification (page 291).
File_name	≜ User_file_name | System_file_name
User_file_name	≜ ’%"’ Manifest string ’%"’
System_file_name	≜ "<" Manifest_string ">"
Signature	≜ "(" Type_list ")" [Result_type]
Type_list	≜ {Type "," ...}
Result_type	≜ ":" Type
Include_files	≜ "|" File_list
File_list	≜ {File_name "," ...}
External_name	≜ **alias** Manifest_string

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

F.3 AVAILABLE POSSIBILITIES

F.3.1 Processing C++ features

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

- If the special feature's declaration simply starts with a C++ class name, followed by the associated file name, it indicates that the Eiffel feature will call a C++ member function (also known as a "method") from that class. The name of the member function 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.

The rest of the possible components are the same as in the C interface: Signature to specify types for arguments and results; possible Include file.

See "AVAILABLE POSSIBILITIES", F.3, page 303.

F.3.2 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*) **is**
        -- Encapsulation of member function *add*.
    external
        "*C++* [*IntArray* %"*intarray.h*%"] (*IntArray* *, *int*)"
    end

This scheme, however, is often inconvenient because it forces the Eiffel side to work on objects in a non-object-oriented way. (The object-oriented 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 the feature

```
add (new_int: INTEGER) is
            -- 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.

F.4 WRAPPING C++ CLASSES: LEGACY++

Before taking a look at examples of the various facilities mentioned, it is useful to consider the tool that will help you, in many cases, avoid worrying about their details.

F.4.1 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 the Legacy++ tool to produce the Eiffel class automatically from the C++ class.

F.4.2 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 in fact the default, so that you do not need to specify –E explicitly (see next option).
- **–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.

F.4.3 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.

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 earlier in this chapter. 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*.

F.4.4 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 — i.e. 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 produce an error (unknown class) at degree 5.

Legacy++ does not handle inline function declarations.

Legacy++ 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.

F.5 AN 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:

```
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++:

indexing

description:
"*Eiffel encapsulation of C++ class IntArray*";
date: "*$Date$*";
revision: "*$Revision$*"

class
INTARRAY

inherit
MEMORY
redefine
dispose
end

creation
make

feature -- Initialization

make (*size*: *INTEGER*) **is**
-- Create Eiffel and C++ objects.
do
object_ptr := *cpp_new* (*size*)
end

```
feature -- Removal

    dispose is
            -- Delete C++ object.
        do
            cpp_delete (object_ptr)
        end

feature

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

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

feature {INTARRAY}

    underscore_integers: POINTER is
            -- 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 %"INTARRAY.H%"] (EIF_INTEGER)"
        end

    cpp_delete (cpp_obj: POINTER) is
            -- Call C++ destructor on C++ object.
        external
            "C++ [delete IntArray %"INTARRAY.H%"] ()"
        end
```

```
cpp_output (cpp_obj: POINTER) is
        -- Call C++ member function.
    external
        "C++ [IntArray %"INTARRAY.H%"] ()"
    alias
        "output"
    end
cpp_add (cpp_obj: POINTER; new_int: INTEGER) is
        -- Call C++ member function.
    external
        "C++ [IntArray %"INTARRAY.H%"] (EIF_INTEGER)"
    alias
        "add"
    end

cpp_underscore_integers (cpp_obj: POINTER): POINTER is
        -- Value of C++ data member
    external
        "C++ [data_member IntArray %"INTARRAY.H%"]: EIF_POINTER"
    alias
        "_integers"
    end

feature {NONE} -- Implementation

    object_ptr: POINTER

end -- class INTARRAY
```