GNATColl: GNAT Reusable Components

Version gpl-2011

AdaCore

Table of Contents

1 Introduction

The GNATColl library provides a number of modules that can be reused in your own applications to add extra features or help implementation.

The modules that are currently provided are:

Script languages
: This module allows you to embed one or more scripting languages in your application, thus providing extensibility to users (see Chapter 3 [Embedding script languages], page 7)

2 Building the GNAT Reusable Components

The compilation process tries to be as flexible as possible. You can choose what modules to build, what features they should have,. . . This flexibility comes at the cost of a certain complexity in the build architecture, but that should be mostly transparent to you.


GNATColl requires a fairly recent Ada05 compatible compiler. If you do not have such a compiler, please contact sales@adacore.com

Since you are reading this documentation, it is assumed you have been able to unpack the package in a temporary directory. In the following instructions, we will assume the following: *prefix* is the directory in which you would like to install GNATColl.

2.1 Configuring the build environment

The first step is to configure the build environment. This is done by running the `configure` command in the root directory of the GNATColl tree.

Some parts of GNATColl need access to a subset of the GNAT sources. This is in particular the case for `GNATCOLL.Projects`, which reuses the same parser as the GNAT tools.

GNATColl will look for those sources in two different ways:

- If you have a copy of the GNAT sources, create a link called '`gnat_src`' that points to the directory containing those sources. This link should be created in the toplevel GNATColl directory.
- Otherwise, recent versions of GNAT come with an additional `gnat_util.gpr` project file, installed along with it. This project contains the required subset of the sources. If you have an older version of GNAT, you could also chose to install `gnat_util` independently.
- If none of the above is satisfied, GNATColl will not include support for `GNATCOLL.Projects`.

`configure` accepts lots of arguments, among which the following ones are most useful:

`--prefix=`*prefix*
: This specifies the directory in which GNATColl should be installed.

`--enable-shared`
`--disable-shared`

If none of these switches is specified, GNATColl will try to build both static and shared libraries (if the latter are supported on your system). The compilation needs to be done twice, since the compilation options might not be the same in both cases.

If you only intend to ever use static libraries, you can use `--disable-shared` to only build static libraries.

When you link GNATColl with your own application, the default is to link with the static libraries. You can change this default, which becomes the shared libraries if you explicitly specify `--enable-shared`. However, even if the default are the static libraries, you can still override that (see below the `LIBRARY_TYPE` variable).

`--with-python=`*`directory`*
`--without-python`

This specifies where GNATColl should find python. If for instance the python executable is in '`/usr/bin`', the *`directory`* to specify is '`/usr`'. In most cases, however, `configure` will be able to detect this automatically, so this is only useful if python is installed in unusual directories. If you specify the second option, support for python will not be build in.

`--enable-shared-python`

This specifies that the python library should be searched directly in *`directory`*/lib, and thus will in general by the shared library. By default, configure will search in a different directory of the python installation, and is more likely to find the static library instead (which makes distributing your application easier). There is no guarantee though that either the shared or the static will be used, since it depends on how python was installed on your system.

`--disable-gtk`

If this switch is specified, then no package depending on the gtk+ graphical toolkit will be built.

`--disable-pygtk`

If this switch is specified, then support for pygtk (see Section 3.1.2 [The Python language], page 10) will not be build. The support for this python module will also be automatically disabled if python was not found or if you configured with `--without-python`.

`--disable-syslog`
: If this switch is specified, then support for syslog (see Section 4.5 [Logging to syslog], page 39) will not be build. This support allows sending the traces from all or part of your application to the system logger, rather than to files or stdout.

`--with-postgresql=<dir>`
`--without-postgresql`
: GNATColl embeds a set of packages to query a database engine. `configure` attempts to find which systems are installed on your system, and build support for those. But you can also explicitly disable for those if you need.

  If the directory in which PostgreSQL is installed contains spaces, you should use a syntax like

  ```
  ./configure --with-postgres="/Program Files/PostgreSQL/8.4"
  ```

  Generally speaking, we do not recommend using paths with spaces since there are often more difficulties in such a setup.

Special support exists in GNATColl for the gtk+ graphical toolkit. `configure` will attempt to find the installation directory for this toolkit by using the `pkg-config` command, which must therefore be available through your *PATH* environment variable. It also needs to find the '`gtkada.gpr`' project file either because it is part of the implicit search path for project files, or because you have put the corresponding directory in the environment variable `GPR_PROJECT_PATH`. If either of these two requirements fail, the modules of GNATColl that depend on GtkAda will not be built.

```
./configure --prefix=/usr/local/gnatcoll --without-python
```

If all goes well (i.e. all required dependencies are found on the system), configure will generate a number of files, including '`Makefile`', '`Makefile.conf`' and '`gnatcoll_shared.gpr`'.

2.2 Building GNATColl

If `configure` has run successfully, it generates a `Makefile` to allow you to build the rest of GNATColl. This is done by simply typing the following command:

```
make
```

Depending on the switches passed to `configure`, this will either build both static and shared libraries, or static only (see the `--disable-shared` configure switch).

Optionally, you can also build the examples and/or the automatic test suite, with the following commands:

```
make examples
```

```
make test
```

The latter will do a local installation of gnatcoll in a subdirectory called '`local_install`', and use this to run the tests. This ensures that the installation process of gnatcoll works properly.

2.3 Installing GNATColl

Installing the library is done with the following command:

```
make install
```

Note that this makefile target does not try to recompile GNATColl, so you must build it first. This will install both the shared and the static libraries if both were build.

As mentioned in the description of the `configure` switches, your application will by default be linked with the static library, unless you specified the `--enable-shared` switch.

However, you can always choose later on which kind of library to use for GNATColl by setting the environment variable `LIBRARY_TYPE` to either `"relocatable"` or `"static"`.

Your application can now use the GNATColl code through a project file, by adding a with clause to '`gnatcoll.gpr`', '`gnatcoll_gtk.gpr`' or '`gnatcoll_python.gpr`'. The second one will also force your application to be linked with the gtk+ libraries, but provides additional capabilities as documented in each of the modules.

3 Embedding script languages

In a lot of contexts, you want to give the possibility to users to extend your application. This can be done in several ways: define an Ada API from which they can build dynamically loadable modules, provide the whole source code to your application and let users recompile it, interface with a simpler scripting languages,. . .

Dynamically loadable modules can be loaded on demand, as their name indicate. However, they generally require a relatively complex environment to build, and are somewhat less portable. But when your users are familiar with Ada, they provide a programming environment in which they are comfortable. As usual, changing the module requires recompilation, re-installation,...

Providing the source code to your application is generally even more complex for users. This requires an even more complex setup, your application is generally too big for users to dive into, and modifications done by one users are hard to provide to other users, or will be lost when you distribute a new version of your application.

The third solution is to embed one or more scripting languages in your application, and export some functions to it. This often requires your users to learn a new language, but these languages are generally relatively simple, and since they are interpreted they are easier to learn in an interactive console. The resulting scripts can easily be redistributed to other users or even distributed with future versions of your application.

The module in GNATColl helps you implement the third solution. It was used extensively in the GPS programming environment for its python interface.

- Each of the scripting language is optional

This module can be compiled with any of these languages as an optional dependency (except for the shell language, which is always built-in, but is extremely minimal, and doesn't have to be loaded at run time anyway). If the necessary libraries are found on the system, GNATColl will be build with support for the corresponding language, but your application can chose at run time whether or not to activate the support for a specific language.

- Optional support is provided for the *gtk+* library

. Likewise, extensions are provided if the gtk+ libraries were found on your system. These provide a number of Ada subprograms that help interface with code using this library, and help export the corresponding classes. This support for gtk+ is also optional, and you can still build GNATColl even if gtk+ wasn't installed on your system (or if your appli-

cation is text-only, in which case you likely do not want to depend at link time on graphical libraries).


Use a scripting language to provide an automatic testing framework for your application.

The GPS environment uses python command for its *automatic test suite*, including graphical tests such as pressing on a button, selecting a menu,. . .

3.1 Supported languages

The module provides built-in support for several scripting languages, and other languages can "easily" be added. Your application does not change when new languages are added, since the interface to export subprograms and classes to the scripting languages is language-neutral, and will automatically export to all known scripting languages.

Support is provided for the following languages:

Shell

This is a very simple-minded scripting language, which doesn't provide flow-control instructions (see Section 3.1.1 [The Shell language], page 8).

Python

Python (`http://www.python.org`) is an advanced scripting language that comes with an extensive library. It is fully object-oriented (see Section 3.1.2 [The Python language], page 10).

3.1.1 The Shell language

The shell language was initially developed in the context of the GPS programming environment, as a way to embed scripting commands in XML configuration files.

In this language, you can execute any of the commands exported by the application, passing any number of arguments they need. Arguments to function calls can, but need not, be quoted. Quoting is only mandatory when they contain spaces, newline characters, or double-quotes ('"'). To quote an argument, surround it by double-quotes, and precede each double-quote it contains by a backslash character. Another way of quoting is similar to what python provides, which is to triple-quote the argument, i.e. surround it by '"""' on each side. In such a case, any special character (in particular other double-quotes or backslashes) lose their special meaning and are just taken as part of the argument. This

is in particular useful when you do not know in advance the contents of the argument you are quoting.

```
Shell> function_name arg1 "arg 2" """arg 3"""
```

Commands are executed as if on a stack machine: the result of a command is pushed on the stack, and later commands can reference it using `%` following by a number. By default, the number of previous results that are kept is set to 9, and this can only be changed by modifying the source code for GNATColl. The return values are also modified by commands executed internally by your application, and that might have no visible output from the user's point of view. As a result, you should never assume you know what `%1`,... contain unless you just executed a command in the same script.

```
Shell> function_name arg1 Shell> function2_name %1
```

In particular, the *%1* syntax is used when emulating object-oriented programming in the shell. A method of a class is just a particular function that contains a '.' in its name, and whose first implicit argument is the instance on which it applies. This instance is generally the result of calling a constructor in an earlier call. Assuming, for instance, that we have exported a class "Base" to the shell from our Ada core, we could use the following code:

```
Shell> Base arg1 arg2 Shell> Base.method %1 arg1 arg2
```

to create an instance and call one of its methods. Of course, the shell is not the best language for object-oriented programming, and better languages should be used instead.

When an instance has associated properties (which you can export from Ada using `Set_Property`), you access the properties by prefixing its name with ":

```
Shell> Base arg1 arg2 # Build new instance Shell> @id %1 # Access its "id
```

Some commands are automatically added to the shell when this scripting language is added to the application. These are

`load` *file* [Function]

Loads the content of *file* from the disk, and execute each of its lines as a Shell command. This can for instance be used to load scripts when your application is loaded

`echo` *`arg...`* [Function]

This function takes any number of argument, and prints them in the console associated with the language. By default, when in an interactive console, the output of commands is automatically printed to the console. But when you execute a script through `load` above, you need to explicitly call `echo` to make some output visible.

`clear_cache` [Function]

This frees the memory used to store the output of previous commands. Calling *`%1`* afterward will not make sense until further commands are executed.

3.1.2 The Python language

Python is an interpreted, object-oriented language. See `http://www.python.org` for more information, including tutorials, on this language.

Python support is optional in GNATColl. If it hasn't been installed on your system, GNATColl will be compiled without it, but that will not impact applications using GNATColl, since the same packages (and the same API therein) are provided in both cases. Of course, if python support wasn't compiled in, these packages will do nothing.

In addition to the API common to all languages (see Section 3.2 [Scripts API], page 13), GNATColl also comes with a low-level interface to the python library. This interface is available in the '`GNATCOLL.Python`' package. In general, it is much simpler to use the common API rather than this specialized one, though, since otherwise you will need to take care of lots of details like memory management, conversion to and from python types,. . .

All functions exported to python are available in a specific namespace

All functions exported to python through GNATColl are available in a single python module, whose name you must specify when adding support for python. This is done to avoid namespace pollution. You can further organize the subprograms through python classes to provide more logical namespaces.

As in Ada, python lets you use named parameters in subprogram calls, and thus let's you change the order of arguments on the command line. This is fully supported by GNATColl, although your callbacks will need to specify the name of the parameters for this to work fine.

```
>>> func_name (arg1, arg2) >>> func_name (arg2=arg2, arg1=arg1)`
```

Some commands and types are always exported by GNATColl, since they are needed by most application, or even internally by GNATColl itself.

`Unexpected_Exception` [Exception]
`Exception` [Exception]
`Missing_Arguments` [Exception]
`Invalid_Argument` [Exception]

A number of exceptions are added automatically, so that the internal state of your application is reflected in python. These are raised on unexpected uncaught Ada exceptions, when your callbacks return explicit errors, or when a function call is missing some arguments.

`exec_in_console` *`command`* [Function]

This function can be used in your script when you need to modify the contents of the python interpreter itself.

When you run a python script, all its commands (including the global variables) are within the context of the script. Therefore, you cannot affect variables which are used for instance in the rest of your application or in the python console. With this function, *`command`* will be executed as if it had been typed in the python console.

```
exec_in_console ("sys.ps1 = 'foo'")
⇒ foo>  # Prompt was changed in the console
```

PyGtk is a python extension that provides an interface to the popular gtk+ library. It gives access to a host of functions for writing graphical interfaces from python. GNATColl interfaces nicely with this extension if it is found.

PyGtk support is also optional. It will be activated in your application if the four following conditions are met: Python was detected on your system, PyGtk was also detected when GNATColl is built, PyGtk is detected dynamically when your application is launched and your code is calling the `Init_PyGtk_Support` function

When PyGtk is detected, you can add the following method to any of the classes you export to python:

`pywidget` [Method on `AnyClass`]

This function returns an instance of a PyGtk class corresponding to the graphical object represented by *`AnyClass`*. In general, it makes

sense when *AnyClass* is bound, in your Ada code, to a GtkAda object. As a result, the same graphical element visible to the user on the screen is available from three different programming languages: C, Ada and Python. All three can manipulate it in the same way

3.1.3 Classes exported to all languages

In addition to the functions exported by each specific scripting language, as described above, GNATColl exports the following to all the scripting languages. These are exported when your Ada code calls the Ada procedure `GNATCOLL.Scripts.Register_Standard_Classes`, which should done after you have loaded all the scripting languages.

`Console` [Class]

`Console` is a name that you can chose yourself when you call the above Ada procedure. It will be assumed to be `Console` in the rest of this document.

This class provides an interface to consoles. A console is an input/output area in your application (whether it is a text area in a graphical application, or simply standard text I/O in text mode). In particular, the python standard output streams `sys.stdin`, `sys.stdout` and `sys.stderr` are redirected to an instance of that class. If you want to see python's error messages or usual output in your application, you must register that class, and define a default console for your scripting language through calls to `GNATCOLL.Scripts.Set_Default_Console`.

You can later add new methods to this class, which would be specific to your application. Or you can derive this class into a new class to achieve a similar goal.

`write` *text* [Method on `Console`]

This method writes *text* to the console associated with the class instance. See the examples delivered with GNATColl for examples on how to create a graphical window and make it into a `Console`.

`clear` [Method on `Console`]

Clears the contents of the console.

`flush` [Method on `Console`]

Does nothing currently, but is needed for compatibility with python. Output through `Console` instances is not buffered anyway.

`Boolean isatty` [Method on `Console`]

Whether the console is a pseudo-terminal. This is always wrong in the case of GNATColl.

`string read [size]` [Method on `Console`]
Reads at most *size* bytes from the console, and returns the resulting string.

`string readline [size]` [Method on `Console`]
Reads at most *size* lines from the console, and returns them as a single string.

3.2 Scripts API

This section will give an overview of the API used in the scripts module. The reference documentation for this API is in the source files themselves. In particular, each '`.ads`' file fully documents all its public API.

As described above, GNATColl contains several levels of API. In particular, it provides a low-level interface to python, in the packages `GNATCOLL.Python`. This interface is used by the rest of GNATColl, but is likely too low-level to really be convenient in your applications, since you need to take care of memory management and type conversions by yourself.

Instead, GNATColl provides a language-neutral Ada API. Using this API, it is transparent for your application whether you are talking to the Shell, to python, or to another language integrated in GNATColl. The code remains exactly the same, and new scripting languages can be added in later releases of GNATColl without requiring a change in your application. This flexibility is central to the design of GNATColl.

In exchange for that flexibility, however, there are language-specific features that cannot be performed through the GNATColl API. At present, this includes for instance exporting functions that return hash tables. But GNATColl doesn't try to export the greatest set of features common to all languages. On the contrary, it tries to fully support all the languages, and provide reasonable fallback for languages that do not support that feature. For instance, named parameters (which are a part of the python language) are fully supported, although the shell language doesn't support them. But that's an implementation detail transparent to your own application.

Likewise, your application might decide to always load the python scripting language. If GNATColl wasn't compiled with python support, the corresponding Ada function still exists (and thus your code still compiles), although of course it does nothing. But since the rest of the code is independent of python, this is totally transparent for your application.


GNATColl comes with some examples, which you can use as a reference when building your own application. See the '`scripts/examples`' directory.

Interfacing your application with the scripting module is a multistep process:

- You *must* **initialize** GNATColl and decide which features to load
- You *can* create an **interactive console** for the various languages, so that users can perform experiments interactively. This is optional, and you could decide to keep the scripting language has a hidden implementation detail (or just for automatic testing purposes for instance)
- You *can* **export** some classes and methods. This is optional, but it doesn't really make sense to just embed a scripting language and export nothing to it. In such a case, you might as well spawn a separate executable.
- You *can* load **start up scripts** or plug-ins that users have written to extend your application.

3.2.1 Initializing the scripting module

GNATColl must be initialized properly in order to provide added value to your application. This cannot be done automatically simply by depending on the library, since this initialization requires multiple-step that must be done at specific moments in the initialization of your whole application.

This initialization does not depend on whether you have build support for python or for gtk+ in GNATColl. The same packages and subprograms are available in all cases, and therefore you do not need conditional compilation in your application to support the various cases.

3.2.1.1 Create the scripts repository

The type `GNATCOLL.Scripts.Scripts_Repository` will contain various variables common to all the scripting languages, as well as a list of the languages that were activated. This is the starting point for all other types, since from there you have access to everything. You will have only one variable of this type in your application, but it should generally be available from all the code that interfaces with the scripting language.

Like the rest of GNATColl, this is a tagged type, which you can extend in your own code. For instance, the GPS programming environment is organized as a kernel and several optional modules. The kernel provides the core functionality of GPS, and should be available from most functions that interface with the scripting languages. Since these functions have very specific profiles, we cannot pass additional arguments to them. One way to work around this limitation is to store the additional arguments (in this case a pointer to the kernel) in a class derived from `Scripts_Repository_Data`.

As a result, the code would look like

```
with GNATCOLL.Scripts;
Repo : Scripts_Repository := new Scripts_Repository_Record;
```

or, in the more complex case of GPS described above:

```
type Kernel_Scripts_Repository is new
Scripts_Repository_Data with record
Kernel : ...;
end record;
Repo : Scripts_Repository := new Kernel_Scripts_Repository'
(Scripts_Repository_Data with Kernel => ...);
```

3.2.1.2 Loading the scripting language

The next step is to decide which scripting languages should be made available to users. This must be done before any function is exported, since only functions exported after a language has been loaded will be made available in that language.

If for instance python support was build into GNATColl, and if you decide not to make it available to users, your application will still be linked with '`libpython`'. It is therefore recommended although not mandatory to only build those languages that you will use

This is done through a simple call to one or more subprograms. The following example registers both the shell and python languages

```
with GNATCOLL.Scripts.Python;
with GNATCOLL.Scripts.Shell;
Register_Shell_Scripting (Repo);
Register_Python_Scripting (Repo, "MyModule");
```

`Register_Shell_Scripting` *Repo* [Procedure]

This adds support for the shell language. Any class or function that is now exported through GNATColl will be made available in the shell

`Register_Python_Scripting` *`Repo Module_Name`* [Procedure]
This adds support for the python language. Any class or function exported from now on will be made available in python, in the module specified by *`Module_Name`*

3.2.1.3 Exporting standard classes

To be fully functional, GNATColl requires some predefined classes to be exported to all languages (see Section 3.1.3 [Classes exported to all languages], page 12). For instance, the `Console` class is needed for proper interactive with the consoles associated with each language.

These classes are created with the following code:

```
Register_Standard_Classes (Repo, "Console");
```

This must be done only after all the scripting languages were loaded in the previous step, since otherwise the new classes would not be visible in the other languages.

`Register_Standard_Classes` *`Repo Console_Class`* [Procedure]
The second parameter *`Console_Class`* is the name of the class that is bound to a console, and thus provides input/output support. You can chose this name so that it matches the classes you intend to export later on from your application.

3.2.2 Creating interactive consoles

The goal of the scripting module in GNATColl is to work both in text-only applications and graphical applications that use the gtk+ toolkit. However, in both cases applications will need a way to capture the output of scripting languages and display them to the user (at least for errors, to help debugging scripts), and possibly emulate input when a script is waiting for such input.

GNATColl solved this problem by using an abstract class `GNATCOLL.Scripts.Virtual_Console_Record` that defines an API for these consoles. This API is used throughout `GNATCOLL.Scripts` whenever input or output has to be performed.


The '`examples/`' directory in the GNATColl package shows how to implement a console in text mode and in graphical mode.

If you want to provide feedback or interact with users, you will need to provide an actual implementation for these `Virtual_Console`, specific to your application. This could be a graphical text window, or based on `Ada.Text_IO`. The full API is fully documented in

'gnatcoll-scripts.ads', but here is a list of the main subprograms that need to be overriden.

`Insert_Text` *`Txt`* [Method on `Virtual_Console`]
`Insert_Log` *`Txt`* [Method on `Virtual_Console`]
`Insert_Error` *`Txt`* [Method on `Virtual_Console`]
These are the various methods for doing output. Error messages could for instance be printed in a different color. Log messages should in general be directed elsewhere, and not be made visible to users unless in special debugging modes.

`Insert_Prompt` *`Txt`* [Method on `Virtual_Console`]
This method must display a prompt so that the user knows input is expected. Graphical consoles will in general need to remember where the prompt ended so that they also know where the user input starts

`Set_As_Default_Console` *`Script`* [Method on `Virtual_Console`]
This method is called when the console becomes the default console for a scripting language. They should in general keep a pointer on that language, so that when the user presses ⟨enter⟩ they know which language must execute the command

`String Read` *`Size Whole_Line`* [Method on `Virtual_Console`]
Read either several characters or whole lines from the console. This is called when the user scripts read from their stdin.

`Set_Data_Primitive` *`Instance`* [Method on `Virtual_Console`]
`Get_Instance` *`Console`* [Method on `Virtual_Console`]
These two methods are responsible for storing an instance of *`Console`* into a `GNATCOLL.Scripts.Class_Instance`. Such an instance is what the user manipulates from his scripting language. But when he executes a method, the Ada callback must know how to get the associated `Virtual_Console` back to perform actual operations on it.

These methods are implemented using one of the `GNATCOLL.Scripts.Set_Data` and `GNATCOLL.Scripts.Get_Data` operations when in text mode, or possibly `GNATCOLL.Scripts.Gtkada.Set_Data` and `GNATCOLL.Scripts.Gtkada.Get_Data` when manipulating graphical GtkAda objects.

There are lots of small details to take into account when writing a graphical console. The example in 'examples/gtkconsole.ads' should provide a good starting point. However, it doesn't handle things like history of commands, preventing the user from moving the cursor to

previous lines,. . . which are all small details that need to be right for the user to feel comfortable with the console.

Once you have created one or more of these console, you can set them as the default console for each of the scripting languages. This way, any input/output done by scripts in this language will interact with that console, instead of being discarded. This is done through code similar to:

```
Console := GtkConsole.Create (...);
Set_Default_Console
(Lookup_Scripting_Language (Repo, "python"),
Virtual_Console (Console));
```

Creating a new instance of `"Console"`, although allowed, will by default create an unusable console. Indeed, depending on your application, you might want to create a new window, reuse an existing one, or do many other things when the user does

```
c = Console()
```

As a result, GNATColl does not try to guess the correct behavior, and thus does not export a constructor for the console. So in the above python code, the default python constructor is used. But this constructor does not associate `c` with any actual `Virtual_Console`, and thus any call to a method of `c` will result in an error.

To make it possible for users to create their own consoles, you need to export a `Constructor_Method` (see below) for the `Console` class. In addition to your own processing, this constructor needs also to call

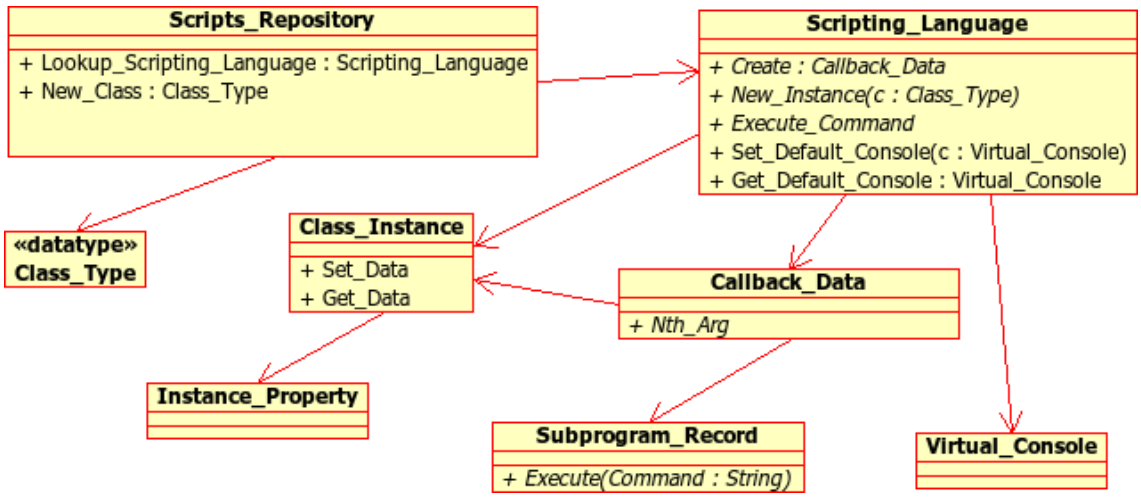
```
declare
Inst : constant Class_Instance := Nth_Arg (Data, 1);
begin
C := new My_Console_Record; -- or your own type
GNATCOLL.Scripts.Set_Data (Inst, C); end
```

3.2.3 Exporting classes and methods

Once all scripting languages have been loaded, you can start exporting new classes and functions to all the scripting languages. It is important to realize that through a single Ada call, they are exported to all loaded scripting languages, without further work required on your part.

3.2.3.1 Classes diagram

The following diagram shows the dependencies between the major data types defined in 'GNATCOLL.Scripts'. Most of these are abstract classes that are implemented by the various scripting languages. Here is a brief description of the role of each type:



`Scripts_Repository` [Class]

As we have seen before, this is a type of which there is a single instance in your whole application, and whose main role is to give access to each of the scripting languages (`Lookup_Scripting_Language` function), and to make it possible to register each exported function only once (it then takes care of exporting it to each scripting language).

`Scripting_Language` [Class]

Instances of this type represent a specific language. It provides various operations to export subprograms, execute commands, create the other types described below,... There should exists a single instance of this class per supported language.

This class interacts with the script interpreter (for instance python), and all code executed in python goes through this type, which then executes your Ada callbacks to perform the actual operation.

It is also associated with a default console, as described above, so that all input and output of the scripts can be made visible to the user.

`Callback_Data` [Class]

This type is an opaque tagged type that provides a language-independent interface to the scripting language. It gives for instance access to the various parameters passed to your subprogram (`Nth_Arg` functions), allows you to set the return value (`Set_Return_Value` procedure), or raise exceptions (`Set_Error_Msg` procedure),. . .

`Class_Type` [Record]

This type is not tagged, and cannot be extended. It basically represents a class in any of the scripting languages, and is used to create new instances of that class from Ada.

`Class_Instance` [Class]

A class instance represents a specific instance of a class. In general, such an instance is strongly bound to an instance of an Ada type. For instance, if you have a `Foo` type in your application that you wish to export, you would create a `Class_Type` called `"Foo"`, and then the user can create as many instances as he wants of that class, each of which is associated with different values of `Foo` in Ada.

Another more specific example is the predefined `Console` class. As we have seen before, this is a `Virtual_Console` in Ada. You could for instance have two graphical windows in your application, each of which is a `Virtual_Console`. In the scripting language, this is exported as a class named `Console`. The user can create two instances of those, each of which is associated with one of your graphical windows. This way, executing `Console.write` on these instances would print the string on their respective graphical window.

Some scripting languages, in particular python, allow you to store any data within the class instances. In the example above, the user could for instance store the time stamp of the last output in each of the instances. It is therefore important that, as much as possible, you always return the same `Class_Instance` for a given Ada object. See the following python example:

```
myconsole = Console ("title") # Create new console
myconsole.mydata = "20060619" # Any data, really
myconsole = Console ("title2") # Create another window
myconsole = Console ("title") # Must be same as first,
print myconsole.mydata # so that this prints "20060619"
```

`Instance_Property` [Class]

As we have seen above, a `Class_Instance` is associated in general with an Ada object. This `Instance_Property` tagged type should be extended for each Ada type you want to be able to store in a `Class_Instance`. You can then use the `Set_Data` and `Get_Data` methods of the `Class_Instance` to get and retrieve that associated Ada object.

`Subprogram_Record` [Class]

This class represents a callback in the scripting language, that is some code that can be executed when some conditions are met.

The exact semantic here depends on each of the programming languages. For instance, if you are programming in python, this is the name of a python method to execute. If you are programming in shell, this is any shell code.

The idea here is to blend in as smoothly as possible with the usual constructs of each language. For instance, in python one would prefer to write the second line rather than the third:

```
def on_exit(): pass
set_on_exit_callback (on_exit) # Yes, python style
set_on_exit_callback ("on_exit") # No
```

The last line (using a string as a parameter) would be extremely unusual in python, and would for instance force you to qualify the subprogram name with the name of its namespace (there would be no implicit namespace resolution).

To support this special type of parameters, the `Subprogram_Record` type was created in Ada.

Although the exact way they are all these types are created is largely irrelevant to your specific application in general, it might be useful for you to override part of the types to provide more advanced features. For instance, GPS redefines its own Shell language, that has basically the same behavior as the Shell language described above but whose `Subprogram_Record` in fact execute internal GPS actions rather than any shell code.

3.2.3.2 Exporting functions

All functions that you export to the scripting languages will result in a call to an Ada subprogram from your own application. This subprogram must have the following profile:

```
procedure Handler
(Data : in out Callback_Data'Class;
Command : String);
```

The first parameter *Data* gives you access to the parameters of the subprogram as passed from the scripting language, and the second parameter *Command* is the name of the command to execute. The idea behind this second parameter is that a single Ada procedure might handle several different script function (for instance because they require common actions to be performed).

`Register_Command` *Repo Command Min_Args Max_Args Handler* [Function]

Each of the shell functions is then exported through a call to `Register_Command`. In its simplest form, this procedure takes the following arguments. *Repo* is the scripts repository, so that the command is exported to all the scripting languages. *Command* is the name of the command. *Min_Args* and *Max_Args* are the minimum and maximum number of arguments. Most language allow option parameters, and this is how you specify them. *Handler* is the Ada procedure to call to execute the command.

Here is a simple example. It implements a function called `Add`, which takes two integers in parameter, and returns their sum.

```
Arg1_C : aliased constant String := "arg1";
Arg2_C : aliased constant String := "arg2";
procedure Sum
(Data : in out Callback_Data'Class;
Command : String)
is
Arg1, Arg2 : Integer;
begin
Name_Parameters ((1 => Arg1_C'Access, 2 => Arg2_C'Access));
Arg1 := Nth_Arg (Data, 1);
Arg2 := Nth_Arg (Data, 2);
Set_Return_Value (Data, Arg1 + Arg2);
end Sum;

Register_Command (Repo, "sum", 2, 2, Sum'Access);
```

Not the most useful function to export! Still, it illustrates a number of important concepts.

Automatic parameters types

When the command is registered, the number of arguments is specified. This means that GNATColl will check on its own whether the right number of arguments is provided. But the type of these arguments is not specified. Instead, your callback should proceed as if they were correct, and try to retrieve them through one of the numerous `Nth_Arg` functions. In the example above, we assume they are integer. But if one of them was passed as a string, an exception would be raised and sent back to the scripting language to display a proper error message to the user. You have nothing special to do here.

Support for named parameters

Some languages (especially python) support named parameters, ie parameters can be specified in any order on the command line, as long as they are properly identified (very similar to Ada's own capabilities). In the example above, the call to `Name_Parameters` is really optional, but adds this support for your own functions as well. You just have to specify the name of the parameters, and GNATColl will then ensure that when you call `Nth_Arg` the parameter number 1 is really `"arg1"`. For scripting languages that do not support named parameters, this has no effect.

Your code can then perform as complex a code as needed, and finally return a value (or not) to the scripting language, through a call to `Set_Return_Value`.

After the above code has been executed, your users can go to the python console and type for instance

```
from MyModule import *    # MyModule is the name we declared above
print sum (1,2)
⇒ 3
print sum ()
 error  Wrong number of parameters
print sum ("1", 2)
 error  Parameter 1 should be an integer
print sum (arg2=2, arg1=1)
⇒ 3
```

3.2.3.3 Exporting classes

Whenever you want to make an Ada type accessible through the scripting languages, you should export it as a class. For object-oriented languages, this would map to the appropriate concept. For other languages, this provides a namespace, so that each method of the class now takes an additional first parameter which is the instance of the class, and the name of the method is prefixed by the class name.

Creating a new class is done through a call to `New_Class`, as shown in the example below.

```
MyClass : Class_Type;
MyClass := GNATCOLL.Scripts.New_Class (Repo, "MyClass");
```

At this stage, nothing is visible in the scripting language, but all the required setup has been done internally so that you can now add methods to this class.

You can then register the class methods in the same way that you registered functions. An additional parameter *`Class`* exists for `Register_Command`. A method is really just a standard function that has an implicit first parameter which is a `Class_Instance`. This extra parameter should not be taken into account in *`Min_Args`* and *`Max_Args`*. You can also declare the method as a static method, ie one that doesn't take this extra implicit parameter, and basically just uses the class as a namespace.

Some special method names are available. In particular, `Constructor_Method` should be used for the constructor of a class. It is a method that receives, as its first argument, a class instance that has just been created. It should associate that instance with the Ada object it represents.

Here is a simple example that exports a class. Each instance of this class is associated with a string, passed in parameter to the constructor. The class has a single method `print`, which prints its string parameter prefixed by the instance's string. To start with, here is a python example on what we want to achieve:

```
c1 = MyClass ("prefix1")
c1.print ("foo")
⇒ "prefix1 foo"
c2 = MyClass ()  # Using a default prefix
c2.print ("foo")
⇒ "default foo"
```

Here is the corresponding Ada code.

```
with GNATCOLL.Scripts.Impl;
procedure Handler
(Data : in out Callback_Data'Class; Command : String)
is
Inst : Class_Instance := Nth_Arg (Data, 1, MyClass);
begin
if Command = Constructor_Method then
Set_Data (Inst, MyClass, Nth_Arg (Data, 2, "default"));
elsif Command = "print" then
Insert_Text
(Get_Script (Data), null,
String'(Get_Data (Inst)) & " " & Nth_Arg (Data, 2));
end if;
end Handler;

Register_Command
(Repo, Constructor_Method, 0, 1, Handler'Access, MyClass);
Register_Command
(Repo, "print", 1, 1, Handler'Access, MyClass);
```

This example also demonstrates a few concepts: the constructor is declared as a method that takes one optional argument. The default value is in fact passed in the call to `Nth_Arg` and is set to `"default"`. In the handler, we know there is always a first argument which is the instance on which the method applies. The implementation for the constructor stores the prefix in the instance itself, so that several instances can have different prefixes (we can't use global variables, of course, since we don't know in advance how many instances will exist). The implementation for `print` inserts code in the default console for the script (we could of course use `Put_Line` or any other way to output data), and computes the string to output by concatenating the instance's prefix and the parameter to `print`.

Note that `Set_Data` and `Get_Data` take the class in parameter, in addition to the class instance. This is needed for proper handling of multiple inheritance: say we have a class `C` that extends two classes `A` and `B`. The Ada code that deals with `A` associates an integer with the class instance, whereas the code that deals with `B` associates a string. Now, if you have an instance of `C` but call a method inherited from `A`, and if `Get_Data` didn't specify the class, there would be a risk that a string would be returned instead of the expected integer. In fact, the proper solution here is that both `A` and `B` store their preferred data at the

same time in the instances, but only fetch the one they actually need. Therefore instances of `C` are associated with two datas.

Here is a more advanced example that shows how to export an Ada object. Let's assume we have the following Ada type that we want to make available to scripts:

```
type MyType is record
Field : Integer;
end record;
```

As you can see, this is not a tagged type, but could certainly be. There is of course no procedure `Set_Data` in 'GNATCOLL.Scripts' that enables us to store `MyType` in a `Class_Instance`. This example shows how to write such a procedure. The rest of the code would be similar to the first example, with a constructor that calls `Set_Data`, and methods that call `Get_Data`.

```
type MyPropsR is new Instance_Property_Record with record
Val : MyType;
end record;
type MyProps is access all MyPropsR'Class;

procedure Set_Data
(Inst : Class_Instance; Val : MyType)
is
begin
Set_Data (Inst, Get_Name (MyClass), MyPropsR'(Val => Val));
end Set_Data;

function Get_Data (Inst : Class_Instance) return MyType is
Data : MyProps := MyProps (Instance_Property'
(Get_Data (Inst, Get_Name (MyClass))));
begin
return Data.Val;
end Get_Data;
```

Several aspects worth noting in this example. Each data is associated with a name, not a class as in the previous example. That's in fact the same thing, and mostly for historical reasons. We have to create our own instance of `Instance_Property_Record` to store the data, but the implementation presents no special difficulty. In fact, we don't absolutely need to create `Set_Data` and `Get_Data` and could do everything inline

in the method implementation, but it is cleaner this way and easier to reuse.

GNATColl is fully responsible for managing the lifetime of the data associated with the class instances and you can override the procedure `Destroy` if you need special memory management.

3.2.3.4 Reusing class instances

We mentioned above that it is more convenient for users of your exported classes if you always return the same class instance for the same Ada object (for instance a graphical window should always be associated with the same class instance), so that users can associate their own internal data with them.

GNATColl provides a few types to facilitate this. In passing, it is worth noting that in fact the Ada objects will be associated with a single instance *per scripting language*, but each language has its own instance. Data is not magically transferred from python to shell!

There are two cases to distinguish here:

- The Ada object derives from a GtkAda object

  In such a case, the package '`GNATCOLL.Scripts.GtkAda`' provides three procedures that automatically associate the instance with the object, and can return the class instance associated with any given GtkAda object, or can return the GtkAda object stored in the instance. There is nothing else to do that to call `Set_Data` as we have seen above. See below for a brief discussion on the Factory design pattern. The internal handling is complex, since python for instance has ref-counted types, and so does gtk+. For the memory to be correctly freed when no longer needed, GNATColl must properly takes care of these reference counting. The result is that the class instance will never be destroyed while the gtk+ object exists, but the gtk+ object might be destroyed while the class instance still exists (in which case no further operation on that instance is possible).

- The Ada object does not derive from a GtkAda object

  In such a case, you should store the list of associated instances with your object. The type `GNATCOLL.Scripts.Instance_List_Access` is meant for that purpose, and provides two `Set` and `Get` primitives to retrieve existing instances.

  There is one catch however, related to memory management. The instances must continue to exist as long as the Ada object exist (and not be destroyed for instance when the python variables goes out of scope). GNATColl mostly takes care of that for you, but requires a little bit of help still: when you implement a new `Instance_Property_Record` as in the example above, you must also override

its primitive `Get_Instances` to return the `Instance_List_Access`. That's it.

The final aspect to consider here is how to return existing instances. This cannot be done from the constructor method, since when it is called it has already received the created instance (this is forced by python, and was done the same for other languages for compatibility reasons). There are two ways to work around that limitation:

- Static `get` methods

  With each of your classes, you can export a static method generally called `get` that takes in parameter a way to identify an existing instance, and either return it or create a new one. It is also recommended to disable the constructor, ie force it to raise an error. Let's examine the python code as it would be used:

```
ed = Editor ("file.adb")  # constructor
⇒ Error, cannot construct instances
ed = Editor.get ("file.adb")
⇒ Create a new instance
ed2 = Editor.get ("file.adb")
⇒ Return existing instance
ed == ed2
⇒ True
```

  The corresponding Ada code would be something like:

```
 type MyType is record
Val : Integer;
Inst : Instance_List_Access;
end record;
type MyTypeAccess is access all MyType;
procedure Handler
(Data : in out Callback_Data'Class; Cmd : String)
is
Inst : Class_Instance;
Tmp : MyTypeAccess;
begin
if Cmd = Constructor_Method then
Set_Error_Msg (Data, "cannot construct instances");
elsif Cmd = "get" then
Tmp := check_if_exists (Nth_Arg (Data, 1));
if Tmp = null then
Tmp := create_new_mytype (Nth_Arg (Data, 1));
Tmp.Inst := new Instance_List;
end if;
Inst := Get (Tmp.Inst.all, Get_Script (Data));
if Inst = null then
Inst := New_Instance (Get_Script (Data), MyClass);
Set (Tmp.Inst.all, Get_Script (Data), Inst);
Set_Data (Inst, Tmp);
end if; return Inst;
end if;
end Handler;
```

- Factory classes

  The standard way to do this in python, which applies to other languages as well, is to use the Factory design pattern. For this, we need to create one class (`MyClassImpl`) and one factory function (`MyClass`).

  The python code now looks like

  ```
  ed = MyClass ("file.adb")  # Create new instance
  ⇒ ed is of type MyClassImpl
  ed = MyClass ("file.adb")  # return same instance
  ed.do_something()
  ```

  It is important to realize that in the call above, we are not calling the constructor of a class, but a function. At the Ada level, the function has basically the same implementation as the one we gave for `get`

above. But the python code looks nicer because we do not have these additional `.get()` calls. The name of the class `MyClassImpl` doesn't appear anywhere in the python code, so this is mostly transparent.

However, if you have more than one scripting language, in particular for the shell, the code looks less nice in this case:

```
MyClass "file.adb"
⇒  <MyClassImpl_Instance_0x12345>
MyClassImpl.do_something %1
```

and the new name of the class is visible in the method call.

3.2.4 Executing startup scripts

The final step in starting up your application is to load extensions or plug-ins written in one of the scripting languages.

There is not much to be said here, except that you should use the `GNATCOLL.Scripts.Execute_File` procedure to do so.

3.2.5 Debugging scripts

GNATColl provides a convenient hook to debug your script. By default, a script (python for instance) will call your Ada callback, which might raise errors. Most of the time, the error should indeed be reported to the user, and you can thus raise a standard exception, or call `Set_Error_Msg`.

BUt if you wish to know which script was executing the command, it is generally not doable. You can however activate a trace (see Chapter 4 [Logging information], page 31) called `"PYTHON.TB"` (for "traceback"), which will output the name of the command that is being executed, as well as the full traceback within the python scripts. This will help you locate which script is raising an exception.

4 Logging information

Most applications need to log various kinds of information: error messages, information messages or debug messages among others. These logs can be displayed and stored in a number of places: standard output, a file, the system logger, an application-specific database table,. . .

The package 'GNATCOLL.Traces' addresses the various needs, except for the application-specific database, which of course is specific to your business and needs various custom fields in any case, which cannot be easily provided through a general interface.

This module is organized around two tagged types (used through access types, in fact, so the latter are mentioned below as a shortcut):

`Trace_Handle`
: This type defines a handle (similar to a file descriptor in other contexts) which is latter used to output messages. An application will generally define several handles, which can be enabled or disabled separately, therefore limiting the amount of logging.

`Trace_Stream`
: Streams are the ultimate types responsible for the output of the messages. One or more handles are associated with each stream. The latter can be a file, the standard output, a graphical window, a socket,. . . New types of streams can easily be defined in your application.

4.1 Configuring traces

As mentioned above, an application will generally create several `Trace_Handle` (typically one per module in the application). When new features are added to the application, the developers will generally need to add lots of traces to help investigate problems once the application is installed at a customer's site. The problem here is that each module might output a lot of information, thus confusing the logs; this also does not help debugging.

The `GNATCOLL.Traces` package allows the user to configure which handles should actually generate logs, and which should just be silent and not generate anything. Depending on the part of the application that needs to be investigated, one can therefore enable a set of handles or another, to be able to concentrate on that part of the application.

This configuration is done at two levels:

- either in the source code itself, where some `trace_handle` might be disabled or enabled by default. This will be described in more details in later sections.
- or in a configuration file which is read at runtime, and overrides the defaults set in the source code.

The configuration file is found in one of three places, in the following order:

- The file name is specified in the source code in the call to `Parse_Config_File`.
- If no file name was specified in that call, the environment variable `ADA_DEBUG_FILE` might point to a configuration file.
- If the above two attempts did not find a suitable configuration file, the current directory is searched for a file called `.gnatdebug`. Finally, the user's home directory will also be searched for that file.

In all cases, the format of the configuration file is the same. Its goal is to associate the name of a `trace_handle` with the name of a `trace_stream` on which it should be displayed.

Streams are identified by a name. You can provide additional streams by creating a new tagged object (see Section 4.4 [Defining custom trace streams], page 38). Here are the various possibilities to reference a stream:

`"name"` where name is a string made of letters, digits and slash ('/') characters. This is the name of a file to which the traces should be redirected. The previous contents of the file is discarded. If the name of the file is a relative path, it is relative to the location of the configuration file, not necessarily to the current directory when the file is parsed. If you used `">>"` instead of `">"` to redirect to that stream, the initial content of the file is not overridden, and new traces are appended to the file instead.

`"&1"` This syntax is similar to the one used on Unix shells, and indicates that the output should be displayed on the standard output for the application. If the application is graphical, and in particular on Windows platforms, it is possible that there is no standard output!

`"&2"` Similar to the previous one, but the output is sent to standard error.

`"&syslog"`
See Section 4.5 [Logging to syslog], page 39.

Comments in a configuration file must be on a line of their own, and start with `--`. Empty lines are ignored. The rest of the lines represent configurations, as in:

- If a line contains the single character `"+"`, it activates all `trace_handle` by default. This means the rest of the configuration file should disable those handles that are not needed. The default is that all handles are disabled by default, and the configuration file should activate the ones it needs. The Ada source code can change the default status of each handles, as well
- If the line starts with the character `">"`, followed by a stream name (as defined above), this becomes the default stream. All handles will be displayed on that stream, unless otherwise specified. If the stream does not exist, it defaults to standard output.
- Otherwise, the first token on the line is the name of a handle. If that is the only element on the line, the handle is activated, and will be displayed on the default stream.

  Otherwise, the next element on the line should be a `"="` sign, followed by either `"yes"` or `"no"`, depending on whether the handle should resp. be enabled or disabled.

  Finally, the rest of the line can optionally contain the `">"` character followed by the name of the stream to which the handle should be directed.

Here is a short example of a configuration file. It activates all handles by default, and defines four handles: two of them are directed to the default stream (standard error), the third one to a file on the disk, and the last one to the system logger syslog (if your system supports it, otherwise to the default stream, ie standard error).

```
 +
>&2
MODULE1
MODULE2=yes
SYSLOG=yes >&syslog:local0:info
FILE=yes >/tmp/file

-- decorators (see below)
DEBUG.COLORS=yes
```

4.2 Using the traces module

If you need or want to parse an external configuration file as described in the first section, the code that initializes your application

should contain a call to `GNATCOLL.Traces.Parse_Config_File`. As documented, this takes in parameter the name of the configuration file to parse. When none is specified, the algorithm specified in the previous section will be used to find an appropriate configuration.

```
GNATCOLL.Traces.Parse_Config_File;
```

The code, as written, will end up looking for a file '`.gnatdebug`' in the current directory.

You then need to declare each of the `trace_handle` that your application will use. The same handle can be declared several times, so the recommended approach is to declare locally in each package body the handles it will need, even if several bodies actually need the same handle. That helps to know which traces to activate when debugging a package, and limits the dependencies of packages on a shared package somewhere that would contain the declaration of all shared handles.

`Trace_Handle Create` *`Name Default Stream Factory Finalize`* [Function]

This function creates (or return an existing) a `trace_handle` with the specified *`Name`*. Its default activation status can also be specified (through *`Default`*), although the default behavior is to get it from the configuration file. If a handle is created several times, only the first call that is executed can define the default activation status, the following calls will have no effect.

`Stream` is the name of the stream to which it should be directed. Here as well, it is generally better to leave things to the configuration file, although in some cases you might want to force a specific behavior.

`Factory` is used to create your own child types of `trace_handle` (see Section 4.3 [Log decorators], page 35).

Here is an example with two package bodies that define their own handles, which are later used for output.

```
package body Pkg1 is
Me : constant Trace_Handle := Create ("PKG1");
Log : constant Trace_Handle := Create ("LOG", Stream => "@syslog");
end Pkg1;
package body Pkg2 is
Me : constant Trace_Handle := Create ("PKG2");
Log : constant Trace_Handle := Create ("LOG", Stream => "@syslog");
end Pkg2;
```

Once the handles have been declared, output is a matter of calling the `GNATCOLL.Traces.Trace` procedure, as in the following sample:

```
Trace (Me, "I am here");
```

Check whether the handle is active

As we noted before, handles can be disabled. In that case, your application should not spend time preparing the output string, since that would be wasted time. In particular, using the standard Ada string concatenation operator requires allocating temporary memory. It is therefore recommended, when the string to display is complex, to first test whether the handle is active. This is done with the following code:

```
if Active (Me) then
Trace (Me, A & B & C & D & E);
end if;
```

An additional subprogram can be used to test for assertions (preconditions or post-conditions in your program), and output a message whether the assertion is met or not.

```
Assert (Me, A = B, "A is not equal to B");
```

If the output of the stream is done in color, a failed assertion is displayed with a red background to make it more obvious.

4.3 Log decorators

Speaking of color, a number of decorators are defined by `GNATCOLL.Traces`. Their goal is not to be used for outputting

information, but to configure what extra information should be output with all log messages. They are activated through the same configuration file as the traces, with the same syntax (i.e either `"=yes"` or `"=no"`).

Here is an exhaustive list:

`DEBUG.ABSOLUTE_TIME`
: If this decorator is activated in the configuration file, the absolute time when Trace is called is automatically added to the output, when the streams supports it (in particular, this has no effect for syslog, which already does this on its own).

`DEBUG.ELAPSED_TIME`
: If this decorator is activated, then the elapsed time since the last call to Trace for the same handle is also displayed.

`DEBUG.STACK_TRACE`
: If this decorator is activated, then the stack trace is also displayed. It can be converted to a symbolic stack trace through the use of the external application `addr2line`, but that would be too costly to do this automatically for each message.

`DEBUG.LOCATION`
: If this decorator is activated, the location of the call to Trace is automatically displayed. This is a file:line:column information. This works even when the executable wasn't compiled with debug information

`DEBUG.ENCLOSING_ENTITY`
: Activate this decorator to automatically display the name of the subprogram that contains the call to `Trace`.

`DEBUG.COLORS`
: If this decorator is activated, the messages will use colors for the various fields, if the stream supports it (syslog doesn't).

`DEBUG.COUNT`
: This decorator displays two additional numbers on each line: the first is the number of times this handle was used so far in the application, the second is the total number of traces emitted so far. These numbers can for instance be used to set conditional breakpoints on a specific trace (break on `gnat.traces.log` or `gnat.traces.trace` and check the value of `Handle.Count`. It can also be used to refer to a specific line in some comment file.

`DEBUG.FINALIZE_TRACES`
: This handle is activated by default, and indicates whether `GNATCOLL.Traces.Finalize` should have any effect. This

can be set to False when debugging, to ensure that traces are available during the finalization of your application.

Here is an example of output where several decorators were activated. In this example, the output is folded on several lines, but in reality everything is output on a single line.

```
 [MODULE] 6/247 User Message (2007-07-03 13:12:53.46)
(elapsed: 2ms)(loc: gnatcoll-traces.adb:224)
(entity:GNATCOLL.Traces.Log)
(callstack: 40FD9902 082FCFDD 082FE8DF )
```

Depending on your application, there are lots of other possible decorators that could be useful (for instance the current thread, or the name of the executable when you have several of them,...). Since `GNATCOLL.Traces` cannot provide all possible decorators, it provides support, through tagged types, so that you can create your own decorators.

This needs you to override the `Trace_Handle_Record` tagged type. Since this type is created through calls to `GNATCOLL.Traces.Create`. This is done by providing an additional *`Factory`* parameter to `Create`; this is a function that allocates and returns the new handle.

Then you can override either (or both) of the primitive operations `Pre_Decorator` and `Post_Decorator`. The following example creates a new type of handles, and prints a constant string just after the module name:

```
 type My_Handle is new Trace_Handle_Record with null record;
procedure Pre_Decorator
(Handle : in out My_Handle;
Stream : in out Trace_Stream_Record'Class;
Message : String) is
begin
Put (Stream, "TEST");
Pre_Decorator (Trace_Handle_Record (Handle), Stream, Message);
end;

function Factory return Trace_Handle is
begin
return new My_Handle;
end;

Me : Trace_Handle := Create ("MODULE", Factory => Factory'Access);
```

As we will see below (see Section 4.6 [Dynamically disabling features], page 40), you can also make all or part of your decorators conditional and configurable through the same configuration file as the trace handles themselves.

4.4 Defining custom trace streams

We noted above that several predefined streams exist, to output to a file, to standard output or to standard error. Depending on your specific needs, you might want to output to other media. For instance, in a graphical application, you could have a window that shows the traces (perhaps in addition to filing them in a file, since otherwise the window would disappear along with its contents if the application crashes); or you could write to a socket (or even a CORBA ORB) to communicate with another application which is charge of monitoring your application.

`GNATCOLL.Traces` provides the type `Trace_Stream_Record`, which can be overridden to redirect the traces to your own streams.

Let's assume for now that you have defined a new type of stream (called `"mystream"`). To keep the example simple, we will assume this stream also redirects to a file. For flexibility, however, you want to let the user configure the file name from the traces configuration file. Here is an example of a configuration file that sets the default stream to a file called '`foo`', and redirects a specific handle to another file called '`bar`'. Note how the same syntax that was used for standard output and standard error is also reused (ie the stream name starts with the `"&"` symbol, to avoid confusion with standard file names).

```
 >&mystream:foo
MODULE=yes >&mystream:bar
```

You need of course to do a bit of coding in Ada to create the stream. This is done by creating a new child of `Trace_Stream_Record`, and override the two primitive operations `Put` and `Newline` (at least). In this implementation, and because `GNATCOLL.Traces.Trace` takes care of not outputting two messages at the same time, we can just output to the file as characters are made available. In some other cases, however, the implementation will need to buffer the characters until the end of line is seen, and output the line with a single call. See for instance the implementation of `GNATCOLL.Traces.Syslog`, which needs to do exactly that.

```
type My_Stream is new Trace_Stream_Record with record
File : access File_Type;
end record;
procedure Put
(Stream : in out My_Stream; Str : String) is
begin
Put (Stream.File.all, Str);
end Put;
procedure Newline (Stream : in out My_Stream) is
begin
New_Line (Stream.File.all);
end Newline;
```

The above code did not open the file itself, as you might have noticed, nor did it register the name `"mystream"` so that it can be used in the configuration file. All this is done by creating a factory, ie a function in charge of creating the new stream. This function receives in parameter the argument specified by the user in the configuration file (after the `":"` character, if any), and must return a newly allocated stream. This function is also never called twice with the same argument, since `GNATCOLL.Traces` automatically reuses an existing stream when one with the same name and arguments already exists.

```
function Factory (Args : String) return Trace_Stream is
Str : access My_Stream := new My_Stream;
begin
Str.File := new File_Type;
Open (Str.File, Out_File, Args);
return Str;
end Factory;

Register_Stream_Factory ("mystream", Factory'Access);
```

4.5 Logging to syslog

Among the predefined streams, GNATColl gives access to the system logger `syslog`. This is a standard utility on all Unix systems, but is not available on other systems. When you compile GNATColl, you should specify the switch `--enable-syslog` to configure to activate the support. If either this switch wasn't specified, or configure could not find the rele-

vant header files anyway, then support for `syslog` will not be available. In this case, the package `GNATCOLL.Traces.Syslog` is still available, but contains a single function that does nothing. If your configuration files redirect some trace handles to `"syslog"`, they will instead be redirect to the default stream or to standard output.

Activating support for syslog requires the following call in your application:

```
GNATCOLL.Traces.Syslog.Register_Syslog_Stream;
```

This procedure is always available, whether your system supports or not syslog, and will simply do nothing if it doesn't support syslog. This means that you do not need to have conditional code in your application to handle that, and you can let GNATColl take care of this.

After the above call, trace handles can be redirected to a stream named `"syslog"`.

The package `GNATCOLL.Traces.Syslog` also contains a low-level interface to syslog, which, although fully functional, you should probably not use, since that would make your code system-dependent.

Syslog itself dispatches its output based on two criteria: the `facility`, which indicates what application emitted the message, and where it should be filed, and the `level` which indicates the urgency level of the message. Both of these criteria can be specified in the `GNATCOLL.Traces` configuration file, as follows:

```
MODULE=yes >&syslog:user:error
```

The above configuration will redirect to a facility called `user`, with an urgency level `error`. See the enumeration types in '`gnatcoll-traces-syslog.ads`' for more information on valid facilities and levels.

4.6 Dynamically disabling features

Although the trace handles are primarily meant for outputting messages, they can be used in another context. The goal is to take advantage of the external configuration file, without reimplementing a similar feature in your application. Since the configuration file can be used to activated or de-activated a handle dynamically, you can then have conditional sections in your application that depends on that handle, as in the following example:

```
CONDITIONAL=yes
```

and in the Ada code:

```
 package Pkg is
Me : constant Trace_Handle := Create ("CONDITIONAL");
begin
if Active (Me) then
... conditional code
end if;
end Pkg;
```

In particular, this can be used if you write your own decorators, as explained above.

5 Monitoring memory

The GNAT compiler allocates and deallocates all memory either through type-specific debug pools that you have defined yourself, or defaults to the standard malloc and free system calls. However, it calls those through an Ada proxy, in the package `System.Memory` that you can also replace in your own application if need be.

`gnatcoll` provides such a possible replacement. Its implementation is also based on `malloc` and `free`, but if you so chose you can activate extra monitoring capabilities to help you find out which parts of your program is allocating the most memory, or where memory is allocated at any moment in the life of your application.

This package is called `GNATCOLL.Memory`. To use it requires a bit of preparation in your application:

- You need to create your own version of '`s-memory.adb`' with the template below, and put it somewhere in your source path. This file should contain the following bit of code

```
 with GNATCOLL.Memory;
package body System.Memory is
package M renames GNATCOLL.Memory;

function Alloc (Size : size_t) return System.Address is
begin
return M.Alloc (M.size_t (Size));
end Alloc;

procedure Free (Ptr : System.Address)
renames M.Free;

function Realloc
(Ptr : System.Address;
Size : size_t)
return System.Address is
begin
return M.Realloc (Ptr, M.size_t (Size));
end Realloc;
end System.Memory;
```

- You then need to compile your application with the extra switch `-a` passed to `gnatmake` or `gprbuild`, so that this file is appropriately compiled and linked with your application

- If you only do this, the monitor is disabled by default. This basically has zero overhead for your application (apart from the initial small allocation of some internal data). When you call the procedure `GNATCOLL.Memory.Configure` to activate the monitor, each memory allocation or deallocation will result in extra overhead that will slow down your application a bit. But at that point you can then get access to the information stored in the monitor

We actually recommend that the activation of the monitor be based on an environment variable or command line switch of your application, so that you can decide at any time to rerun your application with the monitor activated, rather than have to go through an extra recompilation.

All allocations and deallocations are monitor automatically when this module is activated. However, you can also manually call `GNATCOLL.Memory.Mark_Traceback` to add a dummy entry in the internal tables that matches the current stack trace. This is helpful for instance if you want to monitor the calls to a specific subprogram, and know both the number of calls, and which callers executed it how many times. This can help find hotspots in your application to optimize the code.

The information that is available through the monitor is the list of all chunks of memory that were allocated in Ada (this does not include allocations done in other languages like C). These chunks are grouped based on the stack trace at the time of their invocation, and this package knows how many times each stack trace executed each allocation.

As a result, you can call the function `GNATCOLL.Memory.Dump` to dump on the standard output various types of data, sorted. To limit the output to a somewhat usable format, `Dump` asks you to specify how many blocks it should output.

Memory usage
: Blocks are sorted based on the amount of memory they have allocated and is still allocated. This helps you find which part of your application is currently using the most memory.

Allocations count
: Blocks are sorted based on the number of allocation that are still allocated. This helps you find which part of your application has done the most number of allocations (since malloc is a rather slow system call, it is in general a good idea to try and reduce the number of allocations in an application).

Total number of allocations
: This is similar to the above, but includes all allocations ever done in this block, even if memory has been deallocated since then.

Marked blocks

These are the blocks that were created through your calls to `GNATCOLL.Memory.Mark_Traceback`. They are sorted by the number of allocation for that stacktrace, and also shows you the total number of such allocations in marked blocks. This is useful to monitor and analyze calls to specific places in your code

6 Reading and Writing Files

Most applications need to efficiently read files from the disk. Some also need in addition to modify them and write them back. The Ada run-time profiles several high-level functions to do so, most notably in the 'Ada.Text_IO' package. However, these subprograms require a lot of additional housekeeping in the run-time, and therefore tend to be slow.

GNAT provides a number of low-level functions in its 'GNAT.OS_Lib' package. These are direct import of the usual C system calls read(), write() and open(). These are much faster, and suitable for most applications.

However, if you happen to manipulate big files (several megabytes and much more), these functions are still slow. The reason is that to use read you basically need a few other system calls: allocate some memory to temporarily store the contents of the file, then read the whole contents of the file (even if you are only going to read a small part of it, although presumably you would use lseek in such a case).

On most Unix systems, there exists an additional system call mmap() which basically replaces open, and makes the contents of the file immediately accessible, in the order of a few micro-seconds. You do not need to allocate memory specifically for that purpose. When you access part of the file, the actual contents is temporarily mapped in memory by the system. To modify the file, you just modify the contents of the memory, and do not worry about writing the file back to the disk.

When your application does not need to read the whole contents of the file, the speed up can be several orders of magnitude faster than read(). Even when you need to read the whole contents, using mmap() is still two or three times faster, which is especially interesting on big files.

GNATColl's GNATCOLL.Mmap package provides a high-level abstraction on top of the mmap system call. As for most other packages in GNATColl, it also nicely handles the case where your system does not actually support mmap, and will in that case fallback on using read and write transparently. In such a case, your application will perform a little slower, but you do not have to modify your code to adapt it to the new system.

Due to the low-level C API that is needed underneath, the various subprograms in this package do not directly manipulate Ada strings with valid bounds. Instead, a new type Str_Access was defined. It does not contain the bounds of the string, and therefore you cannot use the usual 'First and 'Last attributes on that string. But there are other subprograms that provide those values.

Here is how to read a whole file at once. This is what your code will use in most cases, unless you expect to read files bigger than Integer'Last

bytes long. In such cases you need to read chunks of the file separately. The `mmap` system call is such that its performance does not depend on the size of the file your are mapping. Of course, this could be a problem if `GNATCOLL.Mmap` falls back on calling `read`, since in that case it needs to allocate as much memory as your file. Therefore in some cases you will also want to only read chunks of the file at once.

```
declare
File : Mapped_File;
Str : Str_Access;
begin
File := Open_Read ("/tmp/file_on_disk");
Read (File); -- read the whole file
Str := Data (File);
for S in 1 .. Last (File) loop
Put (Str (S));
end loop;
Close (File);
end;
```

To read only a chunk of the file, your code would look like the following. At the low-level, the system call will always read chunks multiple of a size called the page_size. Although `GNATCOLL.Mmap` takes care of rounding the numbers appropriately, it is recommended that you pass parameters that are multiples of that size. That optimizes the number of system calls you will need to do, and therefore speeds up your application somewhat.

```
 declare
File : Mapped_File;
Str : Str_Access;
Offs : Long_Integer := 0;
Page : constant Integer := Get_Page_Size;
begin
File := Open_Read ("/tmp/file_on_disk");
while Offs < Length (File) loop
Read (File, Offs, Length => Long_Integer (Page) * 4);
Str := Data (File);

– Print characters for this chunk:
for S in Integer (Offs - Offset (File)) + 1 .. Last (File) loop
Put (Str (S));
end loop;

Offs := Offs + Long_Integer (Last (File));
end loop;
Close (File);
```

There are a number of subtle details in the code above. Since the system call only manipulates chunk of the file on boundaries multiple of the code size, there is no guarantee that the part of the file we actually read really starts exactly at *Offs*. If could in fact start before, for rounding issues. Therefore when we loop over the contents of the buffer, we make sure to actually start at the *Offs*-th character in the file.

In the particular case of this code, we make sure we only manipulate multiples of the page_size, so we could in fact replace the loop with the simpler

```
for S in 1 .. Last (File) loop
```

If you intend to modify the contents of the file, not that GNATCOLL.Mmap currently gives you no way to change the size of the file. The only difference compared to the code used for reading the file is the call to open the file, which should be

```
File := Open_Write ("/tmp/file_on_disk");
```

Modifications to Str are automatically reflected in the file. However, there is no guarantee this saving is done immediately. It could be done only when you call `Close`. This is in particular always the case when your system does not support `mmap` and `GNATCOLL.Mmap` had to fallback on calls to `read`.

7 Searching strings

Although the Ada standard provides a number of string-searching subprograms (most notably in the `Ada.Strings.Fixed`, `Ada.Strings.Unbounded` and `Ada.Strings.Bounded` packages through the `Index` functions), these subprograms do not in general provide the most efficient algorithms for searching strings.

The package `GNATCOLL.Boyer_Moore` provides one such optimize algorithm, although there exists several others which might be more efficient depending on the pattern.

It deals with string searching, and does not handle regular expressions for instance.

This algorithm needs to preprocess its key (the searched string), but does not need to perform any specific analysis of the string to be searched. Its execution time can be sub-linear: it doesn't need to actually check every character of the string to be searched, and will skip over some of them. The worst case for this algorithm has been proved to need approximately 3 * N comparisons, hence the algorithm has a complexity of O(n).

The longer the key, the faster the algorithm in general, since that provides more context as to how many characters can be skipped when a non-matching character is found..

We will not go into the details of the algorithm, although a general description follows: when the pattern is being preprocessed, Boyer-Moore computes how many characters can be skipped if an incorrect match is found at that point, depending on which character was read. In addition, this algorithm tries to match the key starting from its end, which in general provides a greater number of characters to skip.

For instance, if you are looking for `"ABC"` in the string `"ABDEFG"` at the first position, the algorithm will compare `"C"` and `"D"`. Since `"D"` does not appear in the key `"ABC"`, it knows that it can immediately skip 3 characters and start the search after `"D"`.

Using this package is extremely easy, and it has only a limited API.

```
declare
Str : constant String := "ABDEABCFGABC";
Key : Pattern;
Index : Integer;
begin
Compile (Key, "ABC");
Index := Search (Key, Str);
end
```

`Search` will either return -1 when the pattern did not match, or the index of the first match in the string. In the example above, it will return 5.

If you want to find the next match, you have to pass a substring to search, as in

```
Index := Search (Key, Str (6 .. Str'Last));
```

8 The templates module

This module provides convenient subprograms for replacing specific substrings with other values. It is typically used to replace substrings like "%{version}" in a longer string with the actual version, at run time.

This module is not the same as the templates parser provided in the context of AWS, the Ada web server, where external files are parsed and processed to generate other files. The latter provides advanced features like filters, loops,. . .

The substrings to be replaced always start with a specific delimiter, which is set to % by default, but can be overridden in your code. The name of the substring to be replaced is then the identifier following that delimiter, with the following rules:

- If the character following the delimiter is the delimiter itself, then the final string will contain a single instance of that delimiter, and no further substitution is done for that delimiter. An example of this is "%%".
- If the character immediately after the delimiter is a curly brace ({), then the name of the identifier is the text until the next closing curly brace. It can then contain any character expect a closing curly brace. An example of this is "%{long name}"
- If the first character after the delimiter is a digit, then the name of the identifier is the number after the delimiter. An example of this is "%12". As a special case, if the first non-digit character is the symbol -, it is added as part of the name of the identifier, as in "%1-". One use for this feature is to indicate you want to replace it with all the positional parameters %1%2%3%4. For instance, if you are writing the command line to spawn an external tool, to which the user can pass any number of parameter, you could specify that command line as "tool -o %1 %2-" to indicate that all parameters should be concatenated on the command line.
- If the first character after the delimiter is a letter, the identifier follows the same rules as for Ada identifiers, and can contain any letter, digit, or underscore character. An example of this is "%ab_12". For readability, it is recommended to use the curly brace notation when the name is complex, but that is not mandatory.
- Otherwise the name of the identifier is the single character following the delimiter

For each substring matching the rules above, the Substitute subprogram will look for possible replacement text in the following order:

- If the Substrings parameter contains an entry for that name, the corresponding value is used.

- Otherwise, if a `callback` was specified, it is called with the name of the identifier, and should return the appropriate substitution (or raise an exception if no such substitution makes sense).
- A default value provided in the substring itself
- When no replacement string was found, the substring is kept unmodified

9 Managing Email

GNATColl provides a set of packages for managing and processing email messages. Through this packages, you can extract the various messages contained in an existing mailbox, extract the various components of a message, editing previously parsed messages, or create new messages from scratch.

This module fully supports MIME-encoded messages, with attachments.

This module currently does not provide a way to send the message through the SMTP protocol. Rather, it is used to create an in-memory representation of the message, which you can then convert to a string, and pass this to a socket. See for instance the AWS library (http://www.adacore.com/home/gnatpro/add-on_technologies/web_technologies) which contains the necessary subprograms to connect with an SMTP server.

9.1 Message formats

The format of mail messages is defined through numerous RFC documents. GNATColl tries to conform to these as best as possible. Basically, a message is made of two parts:

- The headers
  These are various fields that indicate who sent the message, when, to whom, and so on

- The payload (aka body)
  This is the actual contents of the message. It can either be a simple text, or made of one or more attachments in various formats. These attachments can be HTML text, images, or any binary file. Since email transfer is done through various servers, the set of bytes that can be sent is generally limited to 7 bit characters. Therefore, the attachments are generally encoded through one of the encoding defined in the various MIME RFCs, and they need to be decoded before the original file can be manipulated again.

GNATColl gives you access to these various components, as will be seen in the section see Section 9.2 [Parsing messages], page 56.

The package '`GNATCOLL.Email.Utils`' contains various subprograms to decode MIME-encoded streams, which you can use independently from the rest of the packages in the email module.

The headers part of the message contains various pieces of information about the message. Most of the headers have a well-defined semantics and format. However, a user is free to add new headers, which will generally start with `X-` prefix. For those fields where the format is well-defined, they contain various pieces of information:

- Email addresses
  The `From`, `TO` or `CC` fields, among others, contain list of recipients. These recipients are the usual email addresses. However, the format is quite complex, because the full name of the recipient can also be specified, along with comments. The package '`GNATCOLL.Email.Utils`' provides various subprograms for parsing email addresses and list of recipients.
- Dates
  The `Date` header indicates when the message was sent. The format of the date is also precisely defined in the RFC, and the package '`GNATCOLL.Email.Utils`' provides subprograms for parsing this date (or, on the contrary, to create a string from an existing time).
- Text
  The `Subject` header provides a brief overview of the message. It is a simple text header. However, one complication comes from the fact that the user might want to use extended characters not in the ASCII subset. In such cases, the Subject (or part of it) will be MIME-encoded. The package '`GNATCOLL.Email.Utils`' provides subprograms to decode MIME-encoded strings, with the various charsets.

9.2 Parsing messages

There are two ways a message is represented in memory: initially, it is a free-form `String`. The usual Ada operations can be used on the string, of course, but there is no way to extract the various components of the message. For this, the message must first be parsed into an instance of the `Message` type.

This type is controlled, which means that the memory will be freed automatically when the message is no longer needed.

The package '`GNATCOLL.Email.Parser`' provides various subprograms that parse a message (passed as a string), and create a `Message` out of it. Parsing a message might be costly in some cases, for instance if a big attachment needs to be decoded first. In some cases, your application will not need that information (for instance you might only be looking for a few of the headers of the message, and not need any information from the body). This efficiency concern is why there are multiple parsers. Some of them will ignore parts of the message, and thus be more efficient if you can use them.

Once a `Message` has been created, the subprograms in `GNATCOLL.Email` can be used to access its various parts. The documentation for these subprograms is found in the file `gnatcoll-email.ads` directly, and is not duplicated here.

9.3 Parsing mailboxes

Most often, a message is not found on its own (unless you are for instance writing a filter for incoming messages). Instead, the messages are stored in what is called a mailbox. The latter can contain thousands of such messages.

There are traditionally multiple formats that have been used for mailboxes. At this stage, GNATColl only supports one of them, the `mbox` format. In this format, the messages are concatenated in a single file, and separated by a newline.

The package `GNATCOLL.Email.Mailboxes` provides all the types and subprograms to manipulate mailboxes. Tagged types are used, so that new formats of mailboxes can relatively easily be added later on, or in your own application.

Here is a small code example that opens an mbox on the disk, and parses each message it contains

```
declare
Box : Mbox;
Curs : Cursor;
Msg : Message;
begin
Open (Box, Filename => "my_mbox");
Curs := Mbox_Cursor (First (Box));
while Has_Element (Curs) loop
Get_Message (Curs, Box, Msg);
if Msg /= Null_Message then
...
end if;
Next (Curs, Box); end loop;
end;
```

As you can see, the mailbox needs to be opened first. Then we get an iterator (called a cursor, to match the Ada2005 containers naming scheme), and we then parse each message. The `if` test is optional, but recommended: the message that is returned might be null if the mailbox was corrupted and the message could not be parsed. There are

still chances that the next message will be readable, so only the current message should be ignored.

9.4 Creating messages

The subprograms in `GNATCOLL.Email` can also be used to create a message from scratch. Alternatively, if you have already parsed a message, you can alter it, or easily generate a reply to it (using the `Reply_To` subprogram. The latter will preset some headers, so that message threading is preserved in the user's mailers.

10 Ravenscar Patterns

GNATColl provides a set of patterns for concurrent programming using Ravenscar-compliant semantics only. The core goal of the GNATCOLL.Ravenscar (sub) packages is to ease the development of high-integrity multitasking applications by factorizing common behavior into instantiable, Ravenscar-compliant, generic packages. Instances of such generic packages guarantee predictable timing behavior and thus permit the application of most common timing analysis techniques.

10.1 Tasks

The `GNATCOLL.Ravenscar.Simple_Cyclic_Task` generic package lets instantiate a cyclic tasks executing the same operation at regular time intervals; on the other side, the `GNATCOLL.Ravenscar.Simple_Sporadic_Task` task lets instantiate sporadic tasks enforcing a minimum inter-release time.

10.2 Servers

Servers present a more sophisticated run-time semantics than tasks: for example, they can fulfill different kind of requests (see multiple queues servers). `Gnat.Ravenscar.Sporadic_Server_With_Callback` and `Gnat.Ravenscar.Timed_Out_Sporadic_Server` are particularly interesting. The former shows how synchronous inter-task communication can be faked in Ravenscar (the only form of communication permitted by the profile is through shared resources): the server receives a request to fulfill, computes the result and returns it by invoking a call-back. The latter enforces both a minimum and a maximum inter-release time: the server automatically releases itself and invokes an appropriate handler if a request is not posted within a given period of time.

10.3 Timers

`Gnat.Ravenscar.Timers.One_Shot_Timer` is the Ravenscar implementation of time-triggered event through Ada 2005 Timing Events.

11 Managing Memory: The storage pools

Ada gives full control to the user for memory management. That allows for a number of optimization in your application. For instance, if you need to allocate a lot of small chunks of memory, it is generally more efficient to allocate a single large chunk, which is later divided into smaller chunks. That results in a single system call, which speeds up your application.

This can of course be done in most languages. However, that generally means you have to remember not to use the standard memory allocations like `malloc` or `new`, and instead call one of your subprograms. If you ever decide to change the allocation strategy, or want to experiment with several strategies, that means updating your code in several places.

In Ada, when you declare the type of your data, you also specify through a `'Storage_Pool` attribute how the memory for instances of that type should be allocated. And that's it. You then use the usual `new` keyword to allocate memory.

GNATColl provides a number of examples for such storage pools, with various goals. There is also one advanced such pool in the GNAT run-time itself, called `GNAT.Debug_Pools`, which allows you to control memory leaks and whether all accesses do reference valid memory location (and not memory that has already been deallocated).

In GNATColl, you will find the following storage pools:

- `GNATCOLL.Storage_Pools.Alignment`

  This pool gives you full control over the alignment of your data. In general, Ada will only allow you to specify alignments up to a limited number of bytes, because the compiler must only accept alignments that can be satisfied in all contexts, in particular on the stack.

  This package overcomes that limitation, by allocating larger chunks of memory than needed, and returning an address within that chunk which is properly aligned.

12 Manipulating Files

Ada was meant from the beginning to be a very portable language, across architectures. As a result, most of the code you write on one machine has good chances of working as is on other machines. There remains, however, some areas that are somewhat system specific. The Ada run-time, the GNAT specific run-time and GNATColl all try to abstract some of those operations to help you make your code more portable.

One of these areas is related to the way files are represented and manipulated. Reading or writing to a file is system independent, and taken care of by the standard run-time. Other differences between systems include the way file names are represented (can a given file be accessed through various casing or not, are directories separated with a backslash or a forward slash, or some other mean, and a few others). The GNAT run-time does a good job at providing subprograms that work on most types of filesystems, but the relevant subprograms are split between several packages and not always easy to locate. GNATColl groups all these functions into a single convenient tagged type hierarchy. In addition, it provides the framework for transparently manipulating files on other machines.

Another difference is specific to the application code: sometimes, a subprogram needs to manipulate the base name (no directory information) of a file, whereas sometimes the full file name is needed. It is somewhat hard to document this in the API, and certainly fills the code with lots of conversion from full name to base name, and sometimes reverse (which, of course, might be an expansive computation). To make this easier, GNATColl provides a type that encapsulates the notion of a file, and removes the need for the application to indicate whether it needs a full name, a base name, or any other part of the file name.

12.1 Filesystems abstraction

There exists lots of different filesystems on all machines. These include such things as FAT, VFAT, NTFS, ext2, VMS,.... However, all these can be grouped into three families of filesystems:

- windows-based filesystems

  On such filesystems, the full name of a file is split into three parts: the name of the drive (c:, d:,...), the directories which are separated by a backslash, and the base name. Such filesystems are sometimes inaccurately said to be case insensitive: by that, one means that the same file can be accessed through various casing. However, a user is generally expecting a specific casing when a file name is displayed,

and the application should strive to preserve that casing (as opposed to, for instance, systematically convert the file name to lower cases).

A special case of a windows-based filesystems is that emulated by the cygwin development environment. In this case, the filesystem is seen as if it was unix-based (see below), with one special quirk to indicate the drive letter (the file name starts with `"/cygwin/c/"`).

- unix-based filesystems

  On such filesystems, directories are separated by forward slashed. File names are case sensitive, that is a directory can contain both `"foo"` and `"Foo"`, which is not possible on windows-based filesystems.

- vms filesystem

  This filesystem represents path differently than the other two, using brackets to indicate parent directories

A given machine can actually have several file systems in parallel, when a remote disk is mounted through NFS or samba for instance. There is generally no easy way to guess that information automatically, and it generally does not matter since the system will convert from the native file system to that of the remote host transparently (for instance, if you mount a windows disk on a unix machine, you access its files through forward slash- separated directory names).

GNATColl abstracts the differences between these filesystems through a set of tagged types in the `GNATCOLL.Filesystem` package and its children. Such a type has primitive operations to manipulate the names of files (retrieving the base name from a full name for instance), to check various attributes of the file (is this a directory, a symbolic link, is the file readable or writable), or to manipulate the file itself (copying, deleting, reading and writing). It provides similar operations for directories (creating or deleting paths, reading the list of files in a directory,. . .).

It also provides information on the system itself (the list of available drives on a windows machine for instance).

The root type `Filesystem_Record` is abstract, and is specialized in various child types. A convenient factory is provided to return the filesystem appropriate for the local machine (`Get_Local_Filesystem`), but you might chose to create your own factory in your application if you have specialized needs (see Section 12.2 [Remote filesystems], page 65).

12.1.1 file names encoding

One delicate part when dealing with filesystems is handling files whose name cannot be described in ASCII. This includes names in asian languages for instance, or names with accented letters.

There is unfortunately no way, in general, to know what the encoding is for a filesystem. In fact, there might not even be such an encoding (on linux, for instance, one can happily create a file with a chinese name and another one with a french name in the same directory). As a result, GNATColl always treats file names as a series of bytes, and does not try to assume any specific encoding for them. This works fine as long as you are interfacing the system (since the same series of bytes that was returned by it is also used to access the file later on).

However, this becomes a problem when the time comes to display the name for the user (for instance in a graphical interface). At that point, you need to convert the file name to a specific encoding, generally UTF-8 but not necessarily (it could be ISO-8859-1 in some cases for instance).

Since GNATColl cannot guess whether the file names have a specific encoding on the file system, or what encoding you might wish in the end, it lets you take care of the conversion. To do so, you can use either of the two subprograms `Locale_To_Display` and `Set_Locale_To_Display_Encoder`

12.2 Remote filesystems

Once the abstract for filesystems exists, it is tempting to use it to access files on remote machines. There are of course lots of differences with filesystems on the local machine: their names are manipulated similarly (although you need to somehow indicate on which host they are to be found), but any operation of the file itself needs to be done on the remote host itself, as it can't be done through calls to the system's standard C library.

Note that when we speak of disks on a remote machine, we indicate disks that are not accessible locally, for instance through NFS mounts or samba. In such cases, the files are accessed transparently as if they were local, and all this is taken care of by the system itself, no special layer is needed at the application level.

GNATColl provides an extensive framework for manipulating such remote files. It knows what commands need to be run on the remote host to perform the operations (`"cp"` or `"copy"`, `"stat"` or `"dir /a-d"`,...) and will happily perform these operations when you try to manipulate such files.

There are however two operations that your own application needs to take care of to take full advantage of remote files.

12.2.1 Filesystem factory

GNATColl cannot know in advance what filesystem is running on the remote host, so it does not try to guess it. As a result, your application

should have a factory that creates the proper instance of a `Filesystem_Record` depending on the host. Something like:

```
 type Filesystem_Type is (Windows, Unix);
function Filesystem_Factory
(Typ : Filesystem_Type;
Host : String)
return Filesystem_Access
is
FS : Filesystem_Access;
begin
if Host = "" then
case Typ is
when Unix =>
FS := new Unix_Filesystem_Record;
when Windows =>
FS := new Windows_Filesystem_Record;
end case;
else
case Typ is
when Unix =>
FS := new Remote_Unix_Filesystem_Record;
Setup (Remote_Unix_Filesystem_Record (FS.all),
Host => Host,
Transport => ...); -- see below
when Windows =>
FS := new Remote_Windows_Filesystem_Record;
Setup (Remote_Windows_Filesystem_Record (FS.all),
Host => Host,
Transport => ...);
end case;
end if;

Set_Locale_To_Display_Encoder
(FS.all, Encode_To_UTF8'Access);
return FS;
end Filesystem_Factory;
```

12.2.2 Transport layer

There exists lots of protocols to communicate with a remote machine, so as to be able to perform operations on it. These include protocols such as `rsh`, `ssh` or `telnet`. In most of these cases, a user name and password

is needed (and will likely be asked to the user). Furthermore, you might not want to use the same protocol to connect to different machines.

GNATColl does not try to second guess your intention here. It performs all its remote operations through a tagged type defined in `GNATCOLL.Filesystem.Transport`. This type is abstract, and must be overridden in your application. For instance, GPS has a full support for choosing which protocol to use on which host, what kind of filesystem is running on that host, to recognize password queries from the transport protocol,. . . . All these can be encapsulated in the transport protocol.

Once you have created one or more children of `Filesystem_Transport_Record`, you associate them with your instance of the filesystem through a call to the `Setup` primitive operation of the filesystem. See the factory example above.

12.3 Virtual files

As we have seen, the filesystem type abstracts all the operations for manipulating files and their names. There is however another aspect when dealing with file names in an application: it is often unclear whether a full name (with directories) is expected, or whether the base name itself is sufficient. There are also some aspects about a file that can be cached to improve the efficiency.

For these reasons, GNATColl provides a new type `GNATCOLL.VFS.Virtual_File` which abstracts the notion of file. It provides lots of primitive operations to manipulate such files (which are of course implemented based on the filesystem abstract, so support files on remote hosts among other advantages), and encapsulate the base name and the full name of a file so that your API becomes clearer (you are not expecting just any string, but really a file).

This type is reference counted: it takes care of memory management on its own, and will free its internal data (file name and cached data) automatically when the file is no longer needed. This has of course a slight efficiency cost, due to controlled types, but we have found in the context of GPS that the added flexibility was well worth it.

12.4 GtkAda support for virtual files

If you are programming a graphical interface to your application, and the latter is using the `Virtual_File` abstraction all other the place, it might be a problem to convert back to a string when you store a file name in a graphical element (for instance in a tree model if you display an explorer-like interface in your application).

Thus, GNATColl provides the `GNATCOLL.VFS.GtkAda` package, which is only build if `GtkAda` was detected when GNATColl was compiled, which allows you to encapsulate a `Virtual_File` into a `GValue`, and therefore to store it in a tree model.

13 Three state logic

Through the package `GNATCOLL.Tribooleans`, GNATColl provides a type that extends the classical `Boolean` type with an `Indeterminate` value.

There are various cases where such a type is useful. One example we have is when a user is doing a search (on a database or any set of data), and can specify some optional boolean criteria ("must the contact be french?"). He can choose to only see french people ("True"), to see no french people at all ("False"), or to get all contacts ("Indeterminate"). With a classical boolean, there is no way to cover all these cases.

Of course, there are more advanced use cases for such a type. To support these cases, the `Tribooleans` package overrides the usual logical operations `"and"`, `"or"`, `"xor"`, `"not"` and provides an `Equal` function.

See the specs of the package to see the truth tables associated with those operators.

14 Geometry

GNATColl provides the package `GNATCOLL.Geometry`. This package includes a number of primitive operations on geometric figures like points, segments, lines, circles, rectangles and polygons. In particular, you can compute their intersections, the distances,. . .

This package is generic, so that you can specify the type of coordinates you wish to handle.

```
 declare
package Float_Geometry is new GNATCOLL.Geometry (Float);
use Float_Geometry;

P1 : constant Point := (1.0, 1.0);
P2 : constant Point := (2.0, 3.0);
begin
Put_Line ("Distance P1-P2 is" & Distance (P1, P2)'Img);
-- Will print 2.23607
end;
```

Or some operations involving a polygon:

```
 declare
P3 : constant Point := (3.7, 2.0);
P : constant Polygon :=
((2.0, 1.3), (4.1, 3.0), (5.3, 2.6), (2.9, 0.7), (2.0, 1.3));
begin
Put_Line ("Area of polygon:" & Area (P)); -- 3.015
Put_Line ("P3 inside polygon ? " & Inside (P3, P)'Img); -- True
end;
```

15 Reference counting

Memory management is often a difficulty in defining an API. Should we let the user be responsible for freeing the types when they are no longer needed, or can we do it automatically on his behalf ?

The latter approach is somewhat more costly in terms of efficiency (since we need extra house keeping to know when the type is no longer needed), but provides an easier to use API.

Typically, such an approach is implemented using reference counting: all references to an object increment a counter. When a reference disappears, the counter is decremented, and when it finally reaches 0, the object is destroyed.

This approach is made convenient in Ada using controlled types. However, there are a number of issues to take care of to get things exactly right. In particular, the Ada Reference Manual specifies that `Finalize` should be idempotent: it could be called several times for a given object, in particular when exceptions occur.

An additional difficulty is task-safety: incrementing and decrementing the counter should be task safe, since the controlled object might be referenced from several task (the fact that other methods on the object are task safe or not is given by the user application, and cannot be ensures through the reference counting mecanism).

To make things easier, GNATColl provides the package `GNATCOLL.Refcount`. This package contains a generic child package.

To use it, you need to create a new tagged type that extends `GNATCOLL.Refcount.Refcounted`, so that it has a counter. Here is an example.

```
with GNATCOLL.Refcount; use GNATCOLL.Refcount;

package My_Pkg is
type My_Type is new Refcounted with record
Field1 : ...; -- Anything
end record;

package My_Type_Ptr is new Smart_Pointers (My_Type);
end My_Pkg;
```

The code above makes a `Ref` available. This is similar in semantics to an access type, although it really is a controlled type. Every time you

assign the `Ref`, the counter is incremented. When the `Ref` goes out of scope, the counter is decremented, and the object is potentially freed.

Here an example of use of the package:

```
declare
R : Ref;
Tmp : My_Type := ...;
begin
Set (R, Tmp); – Increment counter
Get (R).Field1 := ...; – Access referenced object
end
– R out of scope, so decrement counter, and free Tmp
```

Although reference counting solves most of the issues with memory management, it can get tricky: when there is a cycle between two reference counted objects (one includes a reference to the other, and the other a reference to the first), their counter can never become 0, and thus they are never freed.

There is in particular when common design where this can severly interfer: imagine you want to have a `Map`, associating a name with a reference counted object. Typically, the map would be a cache of some sort. While the object exists, it should be referenced in the map. So we would like the Map to store a reference to the object. But that means the object will then never be freed while the map exists either, and memory usage will only increase.

The solution to this issue is to use `weak references`. These hold a pointer to an object, but do not increase its counter. As a result, the object can eventually be freed. At that point, the internal data in the weak reference is reset to `null`, although the weak reference object itself is still valid.

Here is an example

```
with GNATCOLL.Refcount.Weakref;
use GNATCOLL.Refcount.Weakref;

type My_Type is new Weak_Refcounted with...;

package Pointers is new Weakref_Pointers (My_Type);
```

The above code can be used instead of the code in the first example, and provides the same capability (smart pointers, reference counted

types,...). However, the type `My_Type` is slightly bigger, but can be used to create weak references.

```
 WR : Weak_Ref;

declare
R : Ref;
Tmp : My_Type := ...;
begin
Set (R, Tmp); – Increment counter
WR := Get_Weak_Ref (R); – Get a weak reference

Get (R).Field1 := ...; – Access referenced object
Get (Get (WR)).Field1 := ...; – Access through weak ref
end
– R out of scope, so decrement counter, and free Tmp

if Get (WR) /= Null_Ref then – access to WR still valid
– Always true, since Tmp was freed
end if;
```

The example above is very simplified. Imagine, however, that you store `WR` in a map. Even when `R` is deallocated, the contents of the map remains accessible without a `Storage_Error` (although using `Get` will return `Null_Ref`, as above).

For task-safety issues, `Get` on a weak-reference returns a smart pointer. Therefore, this ensures that the object is never freed while that smart pointer object. As a result, we recommend the following construct in your code:

```
 declare
R : constant Ref := Get (WR);
begin
if R /= Null_Ref then
– Get (R) never becomes null while in this block
end if;
end;
```

16 Configuration files

`gnatcoll` provides a general framework for reading and manipulating configuration files. These files are in general static configuration for your application, and might be different from the preferences that a user might change interactively. However, it is possible to use them for both cases.

There are lots of possible formats for such configuration files: you could chose to use an XML file (but these are in general hard to edit manually), a binary file, or any other format. One format that is found very often is the one used by a lot of Windows applications (the '`.ini`' file format).

`GNATCOLL.Config` is independent from the actual format you are using, and you can add your own parsers compatible with the `GNATCOLL.Config` API. Out of the box, support is provided for '`.ini`' files, so let's detail this very simply format.

```
 # A single-line comment
[Section1]
key1 = value
key2=value2

[Section2]
key1 = value3
```

Comments are (by default) started with `'#'` signs, but you can configure that and use any prefix you want. The `(key, value)` pairs are then organized into optional sections (if you do not start a section before the first key, that key will be considered as part of the `""` section). A section then extends until the start of the next section.

The values associated with the various keys can be strings, integers or booleans. Spaces on the left and right of the values and keys are trimmed, and therefore irrelevant.

Support is providing for interpreting the values as file or directory names. In such a case, if a relative name is specified in the configuration file it will be assumed to be relative to the location of the configuration file (by default, but you can also configure that).

`GNATCOLL.Config` provides an abstract iterator over a config stream (in general, that stream will be a file, but you could conceptually read it from memory, a socket, or any other location). A specific implementation is provided for file-based streams, which is further specialized to parse '`.ini`' files.

Reading all the values from a configuration file is done with a loop similar to:

```
 declare
C : INI_Parser;
begin
Open (C, "settings.txt");
while not At_End (C) loop
Put_Line ("Found key " & Key (C) & " with value " & Value (C));
Next (C);
end loop;
end;
```

This can be made slightly lighter by using the Ada05 dotted notation.

You would only use such a loop in your application if you intend to store the values in various typed constants in your application. But `GNATCOLL.Config` provides a slightly easier interface for this, in the form of a `Config_Pool`. Such a pool is filled by reading a configuration file, and then the values associated with each key can be read at any point during the lifetime of your application. You can also explicitely override the values when needed.

```
 Config : Config_Pool; -- A global variable

declare
C : INI_Parser;
begin
Open (C, "settings.txt");
Fill (Config, C);
end;

Put_Line (Config.Get ("section.key")); -- Ada05 dotted notation
```

Again, the values are by default read as strings, but you can interpret them as integers, booleans or files.

A third layer is provided in `GNATCOLL.Config`. This solves the issue of possible typos in code: in the above example, we could have made a typo when writting `"section.key"`. That would only be detected at run time. Another issue is that we might decide to rename the key in the configuration file. We would then have to go through all the application code to find all the places where this key is references (and that can't be

based on cross-references generated by the compiler, since that's inside a string).

To solve this issue, it is possible to declare a set of constants that represent the keys, and then use these to access the values, solving the two problems above:

```
 Section_Key1 : constant Config_Key := Create ("Key1", "Section");
Section_Key2 : constant Config_Key := Create ("Key2", "Section");

Put_Line (Section_Key1.Get);
```

You then access the value of the keys using the Ada05 dotted notation, providing a very natural syntax. When and if the key is renamed, you then have a single place to change.

17 Projects

The package `GNATCOLL.Projects` provides an extensive interface to parse, manipulate and edit project files ('`.gpr`' files).

Although the interface is best used using the Ada05 notation, it is fully compatible with Ada95.

Here is a quick example on how to use the interface, although the spec file itself contains much more detailed information on all the subprograms related to the manipulation of project files.

```
 with GNATCOLL.Projects; use GNATCOLL.Projects;
with GNATCOLL.VFS; use GNATCOLL.VFS;

Tree : Project_Tree;
Files : File_Array_Access;

Tree.Load (GNATCOLL.VFS.Create (+"path_to_project.gpr"));

– List the source files for project and all imported projects

Files := Tree.Root_Project.Source_Files (Recursive => True);
for F in Files'Range loop
Put_Line ("File is: " & Files (F).Display_Full_Name);
end loop;
```

18 Database interface

GNATColl provides an interface to various database systems. Currently, only PostgreSQL and MySQL are supported, but adding a new back-end is a matter of extending a tagged type and overriding the appropriate subprograms.

This interface was designed with several goals in mind: type-safety, integrity with regards to changes to the database schema, ease of writing queries and performance. A paper was published at the Ada-Europe conference in 2008 which describes the various steps we went through in the design of this library. The rest of this chapter describes the current status of the library, not its history.

18.1 Supported database systems

This library abstracts the specifics of the various database engines it supports. Ideally, a goal written for one database could be ported almost transparently to another engine. This is not completely doable in practice, since each system has its own SQL specifics, and unless you are writing things very carefully, the interpretation of your queries might be different from one system to the next.

However, the Ada code should remain untouched if you change the engine. Various engines are supported out of the box (PostgreSQL and Sqlite), although new ones can be added by overriding the appropriate SQL type (`Database_Connection`). When you compile GNATColl, the build scripts will try and detect what systems are installed on your machine, and only build support for those. It is possible, if no database was installed on your machine at that time, that the database interface API is available (and your application compiles), but no connection can be done to database at run time.

In your code, you will need to create a connection to the database system that you wish to interact with. One possible implementation is

```
 function Connection_Factory
(Desc : GNATCOLL.SQL.Exec.Database_Description)
return GNATCOLL.SQL.Exec.Database_Connection
is
DBMS : constant String := Get_DBMS (Desc);
begin
if DBMS = DBMS_Postgresql then
return GNATCOLL.SQL.Postgres.Build_Postgres_Connection;
elsif DBMS = DBMS_Sqlite then
return GNATCOLL.SQL.Sqlite.Build_Sqlite;
else
return null;
end if;
end Connection_Factory;

declare
DB_Descr : GNATCOLL.SQL.Exec.Database_Description;
DB : GNATCOLL.SQL.Exec.Database_Connection;
begin

GNATCOLL.SQL.Exec.Setup_Database
(Description => DB_Descr,
Database => "dbname");

DB := GNATCOLL.SQL.Exec.Get_Task_Connection
(Description => DB_Descr,
Factory => Connection_Factory'Access,
Username => "myself");
end
```

The code acts in three steps:

- Describe connection parameters The call to `Setup_Database` provides the required parameters to establish a connection to a database server which might possibly be running on a remote host. In this call, one specifies the name of the database, the user login and password, and the type of database. At this point, no connection or exchange of information has been done, the `Database_Description` type stores this information for later.
- Connect to the database The call to `Get_Task_Connection` establishes the actual connection. As will be seen later, the recommended practice when the database backend supports it is to establish one connection per thread in your application, and keep it alive even if

the thread is reused for another reason later on. An example is a web server which has a pool of tasks, and uses the first available one to reply to a request. The function `Get_Task_Connection` will reuse the connection to the database that was already established in this thread, or create a new one through the `Factory` callback if none was created yet.

This call is the only one that is needed in the various tasks of your application, you obviously do not need to repeat the call to `Setup_Database`.

- Create the connection If the current task is not associated with a connection, one needs to be created. GNATColl does this through a factory callback, instead of just doing it on its own, so that your application can create its own child types for a `Database_Connection`, and thus store additional data in that child type. This is also a convenient way to support multiple database backends without `with`-ing all the corresponding code in your application (in the example above, the application only ever supports PostgreSQL, and therefore does not need to elaborate the MySQL packages for instance).

18.2 Database schema monitoring

As stated in the introduction, one of the goals of this library is to make sure the application's code follows changes in the schema of your database. The schema is the list of tables and fields in your database, and the relationship between those tables.

To reach this goal, an external tool, '`gnatcoll_db2ada`' is provided with GNATColl, and should be spawned as the first step of the build process, or at least whenever the database schema changes. It generates an Ada package (`Database`) which reflects the current schema of the database.

This tool supports a number of command line parameters (the complete list of which is available through the '`-h`' switch). The most important of those switches are:

`-dbhost host`
`-dbname name`
`-dbuser user`
`-dbpasswd passwd`
`-dbtype type`

These parameters specify the connection parameters for the database. To find out the schema, '`gnatcoll_db2ada`' needs to connect to that database and do a number of read-only queries. The user does not need to have write permission on the database

`-dbmodel file`

This parameter can replace the above `-dbname`,... It specifies the name of a text file that contains the description of the database, therefore avoiding the need for already having a database up-and-running to generate the Ada interface.

The format of this text file is the same as generated by the `-text` switch (note also that the parser doesn't try to check for all possible types of errors, so if you have a syntax error in your file you might end up with an exception).

This switch is not compatible with `-enum` and `-vars` that really need an access to the database.

`-enum table,id,name,prefix,base`

This parameter can be repeated several times if needed. It identifies one of the special tables of the database that acts as an enumeration type. It is indeed often the case that one or more tables in the database have a role similar to Ada's enumeration types, ie contains a list of values for information like the list of possible priorities, a list of countries,... Such lists are only manipulated by the maintainer of the database, not interactively, and some of their values have impact on the application's code (for instance, if a ticket has an urgent priority, we need to send a reminder every day – but the application needs to know what an urgent priority is). In such a case, it is convenient to generate these values as constants in the generated package. The output will be similar to:

```
 subtype Priority_Id is Integer;
Priority_High : constant Priority_Id := 3;
Priority_Medium : constant Priority_Id := 2;
Priority_Low : constant Priority_Id := 1;
Priority_High_Internal : constant Priority_Id := 4;
```

This code would be extracted from a database table called, for instance, `ticket_priorities`, which contains the following:

```
 table ticket_priorities:
name | priority | category
high | 3 | customer
medium | 2 | customer
low | 1 | customer
high_internal | 4 | internal
```

To generate the above Ada code, you need to pass the following parameter to 'gnatcoll_db2ada':

```
-enum ticket_priorities,Priority,Priority,Integer
```

where the second parameter is the name of the field in the table, and the first is the prefix to add in front of the name to generate the Ada constant's name. The last parameter should be either `Integer` or `String`, which influences the way the value of the Ada constant is generated (surrounded or not by quotes).

`-var name,table,field,criteria,comment`

This is similar to the `-enum` switch, but extracts a single value from the database. Although applications should try and depend as little as possible on such specific values, it is sometimes unavoidable.

For instance, if we have a table in the table with the following contents:

```
 table staff
staff_id | login
0 | unassigned
1 | user1
```

We could extract the id that helps detect unassigned tickets with the following command line:

```
-var no_assign_id,staff,staff_id,"login='unassigned'","help"
```

which generates

```
No_Assigne_Id : constant := 0;
-- help
```

The application should use this constant rather than some hard-coded string `"unassigned"` or a named constant with the same value. The reason is that presumably the login will be made visible somewhere to the user, and we could decide to change it (or translate it to another language). In such a case, the application would break. On the other hand, using

the constant `0` which we just extracted will remain valid, whatever the actual text we display for the user.

`-text`

Instead of creating Ada files to represent the database schema, this switch will ask `gnatcoll_db2ada` to dump the schema as text. This is in a form hopefully easy to parse automatically, in case you have tools that need the schema information from your database in a DBMS-independent manner. See below for a description of the format.

`-createdb`

Instead of the usual default output, `gnatcoll_db2ada` will output a set of SQL commands that can be used to re-create the set of all tables in your schema. This does not create the database itself (which might require special rights depending on your DBMS), only the tables.

18.2.1 Textual description of database schema

`gnatcoll_db2ada` can either automatically extracts the database schema from a running instance of your DBMS (see `-dbname`), or get the schema from a text file (see `-dbmodel`). In the latter case, it can then create the database for you.

This text file is better since you have more control over names (for instance for foreign keys) and is slightly higher level.

This file is a collection of paragraphs, each of which relates to one table or one SQL view in your database. The paragraphs start with a line containing:

```
 table ::= '|' ('ABSTRACT')? ('TABLE'|'VIEW') ['(' supertable ')']
'|' <name> '|' <name_row>
```

"name" is the name of the table. The third pipe and third column are optional, and should be used to specify the name for the element represented by a single row. For instance, if the table is called "books", the third column could contain "book". This is mostly used when generating high-level Ada code, an upcoming feature not yet available in GNATCOLL.

If the first line starts with the keyword `ABSTRACT`, then no instance of that table actually exists in the database. This is used in the context of table inheritance, so define shared fields only once among multiple tables.

The keyword `TABLE` can be followed by the name of a table from which it inherits the fields. Currently, that supertable must be abstract, and the fields declared in that table are simply duplicated in the new table. For instance:

```
 | ABSTRACT TABLE | parent | | |
| field1 | INTEGER | | |

| TABLE mytable(parent) | | | | |
| id | INTEGER | PK | | |
```

The table `mytable` in fact has two columns, `id` and `field1`.

Each line must then start with a pipe character (`"|"`), and contain a number of pipe-separated fields. The order of the fields is always given by the following grammar:

```
 fields ::= '|' <name> '|' <type>
'|' ('PK'|''|'NULL'|'NOT NULL'|'INDEX') '|' [default] '|' [doc] '|'
```

The type of the fields is the SQL type (`"INTEGER"`, `"TEXT"`,...). The tool will automatically convert these to Ada when generating Ada code. A special type (`"AUTOINCREMENT"`) is an integer that is automatically incremented according to available ids in the table. The exact type used will depend on the specific DBMS.

If the field is a foreign key (that is a value that must correspond to a row in another table), you can use the special syntax

```
fk_type ::= 'FK' <table_name> (<revert_name>)
```

As you can see, the type of the field is not specified explicitly, but will always be that of the foreign table's primary key. With this syntax, the foreign table must have a single field for its primary key.

In the future, gnatcoll will include a tool that generates an Ada API that completly hides the underlying SQL commands. The `"revert_name"` argument is used in the context of that tool. Although you are encouraged to provide one, this is currently optional and will have no impact on the output of the tool.

`"revert_name"` is the name that will be generated in the Ada code for the reverse relationship. For instance: assume a book is always found in a library. If we have two tables `"books"` and `"libraries"`, we would

have one foreign key from "books" to "libraries", which, for each book, indicates to which library it belongs. As a result, in the generated code, it would be easy to use "mybook.Library" to get that library. But given a library, we also want to be able to retrieve all its books with code similar to "mylibrary.Books". The foreign key would be declared as such in the books table:

```
table books | library | FK libraries(books) | NOT NULL | | where the book
```

If the "revert_name" is empty (the parenthesis are shown), no revert relationship is generated. If the parenthesis and the revert_name are both omitted, a default name is generated.

The third column in the fields definition indicates whether we have a primary key ("PK"), which must never be null. In other cases, the column indicates whether the column can have null values (the default, or explicitly "NULL"), or not ("NOT NULL"). If "INDEX" is specified, an index will be created for that particular column. These are similar to the corresponding SQL constraints.

Multiple keywords can be used if they are separated by commas. Thus, "NOT NULL, INDEX" indicates a column that must be set by the user, and for which an index is created to speed up look ups.

The fourth column gives the default value for the field, and is given in SQL.

The fifth column contains documentation for the field (if any). This documentation will be included in the generated code, so that IDEs can provide useful tooltips when navigating your application's code.

After all the fields have been defined, you can specify extract constraints on the table. In particular, if you have a foreign key to a table that uses a tuple as its primary key, you can define that foreign key as

```
FK ::= '|' "FK:" '|' <table> '|' <field_names>* '|' <field_names>* '|'

table tableA | FK: | tableB | fieldA1, fieldA2 | fieldB1, fieldB2 |
```

18.2.2 Default output of gnatcoll_db2ada

From the command line arguments, `gnatcoll_db2ada` will generate an Ada package, which contains one type per table in the database. Each of these types has a similar structure. The implementation details are not shown here, since they are mostly irrelevant and might change. Currently, a lot of this code are types with discriminants. The latter are

`access-to-string`, to avoid duplicating strings in memory and allocating and freeing memory for these. This provides a better performance.

```
package Database is
type T_Ticket_Priorities (...) is new SQL_Table (...) with record
Priority : SQL_Field_Integer;
Name : SQL_Field_Text;
end record;

overriding function FK (Self : T_Ticket_Priorities; Foreign : SQL_Table'Cl
return SQL_Criteria;

Ticket_Priorities : constant T_Ticket_Priorities (...);
end Database;
```

It provides a default instance of that type, which can be used to write queries (see the next section). This type overrides one primitive operation which is used to compute the foreign keys between that table and any other table in the database (see Section 18.3 [Writing queries], page 91).

Note that the fields which are generated for the table (our example reuses the previously seen table `ticket_priorities`) are typed, which as we will see provides a simple additional type safety for our SQL queries.

18.3 Writing queries

The second part of the database support in GNATColl is a set of Ada subprograms which help write SQL queries. Traditional ways to write such queries have been through embedded SQL (which requires a preprocessing phase and complicate the editing of source files in Ada-aware editors), or through simple strings that are passed as is to the server. In the latter case, the compiler can not do any verification on the string, and errors such a missing parenthesis or misspelled table or field names will not be detected until the code executes the query.

GNATColl tries to make sure that code that compiles contains syntactically correct SQL queries and only reference existing tables and fields. This of course does not ensure that the query is semantically correct, but helps detect trivial errors as early as possible.

Such queries are thus written via calls to Ada subprograms, as in the following example.

```
 with GNATCOLL.SQL; use GNATCOLL.SQL;
with Database; use Database;
declare
Q : SQL_Query;
begin
Q := SQL_Select
(Fields => Max (Ticket_Priorities.Priority)
& Ticket_Priorities.Category,
From => Ticket_Priorities,
Where => Ticket_Priorities.Name /= "low",
Group_By => Ticket_Priorities.Category);
end
```

The above example will return, for each type of priority (internal or customer) the highest possible value. The interest of this query is left to the user...

This is very similar to an actual SQL query. Field and table names come from the package that was automatically generated by the `gnatcoll_db2ada` tool, and therefore we know that our query is only referencing existing fields. The syntactic correctness is ensured by standard Ada rules. The `SQL_Select` accepts several parameters corresponding to the usual SQL attributes like `GROUP BY`, `HAVING`, `ORDER BY` and `LIMIT`.

The `From` parameter could be a list of tables if we need to join them in some ways. Such a list is created with the overridden `"&"` operator, just as for fields which you can see in the above example. GNATColl also provides a `Left_Join` function to join two tables when the second might have no matching field (see the SQL documentation).

Similar functions exist for `SQL_Insert`, `SQL_Update` and `SQL_Delete`. Each of those is extensively documented in the '`gnatcoll-sql.ads`' file.

It is worth noting that we do not have to write the query all at once. In fact, we could build it depending on some other criteria. For instance, imagine we have a procedure that does the query above, and omits the priority specified as a parameter, or shows all priorities if the empty string is passed. Such a procedure could be written as

```
 procedure List_Priorities (Omit : String := "") is
Q : SQL_Query;
C : SQL_Criteria := No_Criteria;
begin
if Omit /= "" then
C := Ticket_Priorities.Name /= Omit;
end if;
Q := SQL_Select
(Fields => ..., -- as before
Where => C);
end;
```

With such a code, it becomes easier to create queries on the fly than it would be with directly writing strings.

The above call has not sent anything to the database yet, only created a data structure in memory (more precisely a tree). In fact, we could be somewhat lazy when writing the query and rely on auto-completion, as in the following example:

```
 Q := SQL_Select
(Fields => Max (Ticket_Priorities.Priority)
& Ticket_Priorities.Category,
Where => Ticket_Priorities.Name /= "low");

Auto_Complete (Q);
```

This query is exactly the same as before. However, we did not have to specify the list of tables (which GNATColl can compute on its own by looking at all the fields referenced in the query), nor the list of fields in the `GROUP BY` clause, which once again can be computed automatically by looking at those fields that are not used in a SQL aggregate function. This auto-completion helps the maintenance of those queries.

There is another case where GNATColl makes it somewhat easier to write the queries, and that is to handle joins between tables. If your schema was build with foreign keys, GNATColl can take advantage of those.

For instance, imagine we have a table `ticket`. These tickets have a priority, which is an integer id pointing to the `ticket_priorities` table we saw previously. In SQL, this means that the `ticket` table has a foreign key on the `ticket_priorities` table, which implies that any priority set for a ticket must exist in the second table. Using an id instead of the actual name of the priority in the `ticket` table means that

it is easy to change the name of the priority, without impacting the rest of the database.

Imagine, now, that a query needs to list all tickets with their priorities. Since we want to show the output to the user, we do not want to show the internal id, but the actual name of the priority. This would be done with a query similar to:

```
 Q := SQL_Select
(Fields => Ticket.Number & Ticket_Priorities.Name,
From => Ticket & Ticket_Priorities,
Where => Ticket.Priority = Ticket_Priorities.Priority);
```

In fact, with the auto-completion, we could write it as

```
 Q := SQL_Select
(Fields => Ticket.Number & Ticket_Priorities.Name,
Where => Ticket.Priority = Ticket_Priorities.Priority);
```

The `WHERE` clause comes straight from the definition of the foreign key. It can be shorten using the `FK` primitive operation that we saw was generated in the `Database` package. The following example uses the Ada05 dotted notation for the call to FK, but that is not mandatory, of course.

```
 Q := SQL_Select
(Fields => Ticket.Number & Ticket_Priorities.Name,
Where => Ticket.FK (Ticket_Priorities));
```

One advantage is that we avoid possible errors in writing the join, and if at some point the foreign key between the two tables involves more fields, for instance, the query remains valid and we do not have to change our code.

18.4 Executing queries

Once we have our query in memory, we need to pass it on to the database server itself, and retrieve the results.

Executing is done through the `GNATCOLL.SQL.Exec` package, as in the following example:

```
declare
R : Forward_Cursor;
begin
Execute (Connection => DB, Result => R, Query => Q); end;
```

This reuses the connection we have established previously (`DB`), and sends it the query. The result of that query is then stored in `R`, to be used later.

There are several versions of `Execute`. In particular, there are versions which do not have the `R` parameter. Some queries do not return anything useful to our application (for instance a `INSERT` query), and thus we can simplify the call.

`Execute` has an extra parameter `Use_Cache`, set to `False` by default. If this parameter is true, and the exact same query has already been executed before, its result will be reused without even contacting the database server. The cache is automatically invalidated every hour in any case. This cache is mostly useful for tables that act like enumeration types, as we have seen before when discussing the `-enum` parameter to '`gnatcoll_db2ada`'. In this case, the contents of the table changes very rarely, and the cache can provide important speedups, whether the server is local or distant. However, we recommend that you do actual measurements to know whether this is indeed beneficial for you. You can always invalidated the current cache with a call to `Invalidate_Cache` to force the query to be done on the database server.

If for some reason the connection to the database is no longer valid (a transient network problem for instance), GNATColl will attempt to reconnect and re-execute your query transparently, so that your application does not need to handle this case.

If your query produces an error (whether it is invalid, or any other reason), a flag is toggled in the `Connection` parameter, which you can query through the `Success` subprogram. As a result, a possible continuation of the above code is:

```
if Success (DB) then
...
else
... an error occurred
end if
```

GNATColl also tries to be helpful in the way it handles SQL transactions. Such transactions are a way to execute your query in a sandbox,

ie without affecting the database itself until you decide to `COMMIT` the query. Should you decide to abort it (or `ROLLBACK` as they say for SQL), then it is just as if nothing happened. As a result, it is in general recommended to do all your changes to the database from within a transaction. If one of the queries fail because of invalid parameters, you just rollback and report the error to the user. The database is still left in a consistent state. As an additional benefit, executing within a transaction is sometimes faster, as is the case for PostgreSQL for instance.

To help with this, GNATColl will automatically start a transaction the first time you edit the database. It is then your responsibility to either commit or rollback the transaction when you are done modifying. A lot of database engines (among which PostgreSQL) will not accept any further change to the database if one command in the transaction has failed. To take advantage of this, GNATColl will therefore not even send the command to the server if it is in a failure state.

Here is code sample that modifies the database:

```
 Execute (DB, SQL_Insert (...));
– The code above starts a transaction and inserts a new row

Execute (DB, SQL_Insert (...));
– Executed in the same transaction

Commit_Or_Rollback (DB);
– Commit if both insertion succeeded, rollback otherwise
– You can still check Success(DB) afterward if needed
```

18.5 Prepared queries

The previous section showed how to execute queries and statements. But these were in fact relatively inefficient.

With most DBMS servers, it is possible to compile the query once on the server, and then reuse that prepared query to significantly speed up later searches when you reuse that prepared statement.

It is of course pretty rare to run exactly the same query or statement multiple times with the same values. For instance, the following query would not give much benefit if it was prepared, since you are unlikely to reuse it exactly as is later on.

```
SELECT * FROM data WHERE id=1
```

SQL (and GNATColl) provide a way to parameterize queries. Instead of hard-code the value `1` in the example above, you would in fact use a special character (unfortunately specific to the DBMS you are interfacing to) to indicate that the value will be provided when the query is actually executed. For instance, `sqlite` would use:

```
SELECT * FROM data WHERE id=?
```

You can write such a query in a DBMS-agnostic way by using GNATColl. Assuming you have automatically generated '`database.ads`' by using `gnatcoll_db2ada`, here is the corresponding Ada code:

```
with Database; use Database;

Q : constant SQL_Query :=
SQL_Select
(Fields => Data.Id & Data.Name
From => Data,
Where => Data.Id = Integer_Param (1));
```

GNATColl provides a number of functions (one per type of field) to indicate that the value is currently unbound. `Integer_Param`, `Text_Param`, `Boolean_Param`,... all take a single argument, which is the index of the corresponding parameter. A query might need several parameters, and each should have a different index. On the other hand, the same parameter could be used in several places in the query.

Although the query above could be executed as is by providing the values for the parameters, it is more efficient, as we mentioned at the beginning, to compile it on the server. In theory, this preparation is done within the context of a database connection (thus cannot be done for a global variable, where we do not have connections yet, and where the query might be executed by any connection later on).

GNATColl will let you indicate that the query should be prepared. This basically sets up some internal data, but does not immediately compile it on the server. The first time the query is executed in a given connection, though, it will first be compiled. The result of this compilation will be reused for that connection from then on. If you are using a second connection, it will do its own compilation of the query.

So in our example we would add the following global variable:

```
 P : constant Prepared_Statement :=
Prepare (Q, On_Server => True);
```

Two comments about this code:

- You do not have to use global variables. You can prepare the statement locally in a subprogram. A `Prepared_Statement` is a reference counted type, that will automatically free the memory on the server when it goes out of scope.
- Here, we prepared the statement on the server. If we had specified `On_Server => False`, we would still have sped things up, since Q would be converted to a string that can be sent to the DBMS, and from then on reused that string (note that this conversion is specific to each DBMS, since they don't always represent things the same way, in particular parameters, as we have seen above).

Now that we have a prepared statement, we can simply execute it. If the statement does not require parameters, the usual `Fetch` and `Execute` subprograms have versions that work exactly the same with prepared statements. They also accept a `Params` parameter that contains the parameter to pass to the server. A number of `"+"` operators are provided to create those parameters.

```
 declare
F : Forward_Cursor;
begin
F.Fetch (DB, P, Params => (1 => +2));
F.Fetch (DB, P, Params => (1 => +3));
end;
```

Note that for string parameters, the `"+"` operator takes an access to a string. This is for efficiency, to avoid allocating memory and copying the string, and is safe because the parameters are only needed while `Fetch` executes (even for a `Forward_Cursor`.

There is one last property on `Prepared_Statement`s: when you prepare them, you can pass a `Use_Cache => True` parameter. When this is used, the result of the query will be cached by GNATColl, and reuse when the query is executed again later. This is the fastest way to get the query, but should be used with care, since it will not detect changes in the database. The local cache is automatically invalidated every hour, so the query will be performed again at most one hour later. Local caching

is disabled when you execute a query with parameters. In this case, prepare the query on the server which will still be reasonably fast.

Finally, here are some examples of timings. The exact timing are irrelevant, but it is interesting to look at the different between the various scenarios. Each of them performs 100_000 simple queries similar to the one used in this section.

```
 Not preparing the query, using Direct_Cursor:
4.05s

Not preparing the query, using Forward_Cursor, and only
retrieving the first row:
3.69s

Preparing the query on the client (On_Server => False),
with a Direct_Cursor. This saves the whole GNATCOLL.SQL
manipulations and allocations:
2.50s

Preparing the query on the server, using Direct_Cursor:
0.55s

Caching the query locally (Use_Cache => True):
0.13s
```

18.6 Getting results

One you have executed a `SELECT` query, you generally need to examine the rows that were returned by the database server. This is done in a loop, as in

```
 while Has_Row (R) loop
Put_Line ("Max priority=" & Integer_Value (R, 0)'Img
& " for category=" & Value (R, 1));
Next (R);
end loop;
```

You can only read one row at a time, and as soon as you have moved to the next row, there is no way to access a previously fetched row. This is the greatest common denominator between the various database

systems. In particular, it proves efficient, since only one row needs to be kept in memory at any point in time.

For each row, we then call one of the `Value` or `*Value` functions which return the value in a specific row and a specific column.

18.7 Writing your own cursors

The cursor interface we just saw is low-level, in that you get access to each of the fields one by one. Often, when you design your own application, it is better to abstract the database interface layer as much as possible. As a result, it is often better to create record or other Ada types to represent the contents of a row.

Fortunately, this can be done very easily based on the low-level interface provided by GNATCOLL. Here is a code example that shows how this can be done.

```
type My_Row is record
Id : Integer;
Name : Unbounded_String;
end record;

type My_Cursor is new Forward_Cursor with null record;
function Element (Self : My_Cursor) return My_Row;
function Do_Query return My_Cursor;
```

The idea is that you create a function that does the query for you (based on some parameters that are not show here), and then returns a cursor over the resulting set of rows. For each row, you can use the `Element` function to get an Ada record for easier manipulation.

Let's first see how these types would be used in practice:

```
declare
C : My_Cursor := Do_Query (...);
begin
while Has_Row (C) loop
Put_Line ("Id = " & Element (C).Id);
Next (C);
end loop;
end;
```

So the loop itself is the same as before, except we no longer access each of the individual fields directly. This means that if the query changes to return more fields (or the same fields in a differente order for instance), the code in your application does not need to change.

The specific implementation of the subprograms could be similar to the following subprograms (we do not detail the writing of the SQL query itself, which of course is specific to your application)

```
function Do_Query return My_Cursor is
Q : constant SQL_Query := ....;
R : My_Cursor;
begin
Execute (DB, R, Q);
return R;
end Do_Query;

function Element (Self : My_Cursor) return My_Row is
begin
return My_Row'
(Id => Integer_Value (Self, 0),
Name => To_Unbounded_String (Value (Self, 1)));
end Element;
```

There is one more complex case though. It might happen that an element needs access to several rows to fill the Ada record. For instance, if we are writing a CRM application and query the contacts and the companies they work for, it is possible that a contact works for several companies. The result of the SQL query would then look like this:

```
 contact_id | company_id
1 | 100
1 | 101
2 | 100
```

The sample code shown above will not work in this case, since Element is not allowed to modify the cursor. In such a case, we need to take a slightly different approach.

```
type My_Cursor is new Forward_Cursor with null record;
function Do_Query return My_Cursor; -- as before
procedure Element_And_Next
(Self : in out My_Cursor; Value : out My_Row);
```

where `Element_And_Next` will fill Value and call Next as many times as needed. On exit, the cursor is left on the next row to be processed. The usage then becomes

```
while Has_Row (R) loop
Element_And_Next (R, Value);
end loop;
```

To prevent the user from using Next incorrectly, you should probably override `Next` with a procedure that does nothing (or raises a Program_Error maybe). Make sure that in `Element_And_Next` you are calling the inherited function, not the one you have overridden, though.

There is still one more catch. The user might depend on the two subprograms `Rows_Count` and `Processed_Rows` to find out how many rows there were in the query. In practice, he will likely be interested in the number of distinct contacts in the tables (2 in our example) rather than the number of rows in the result (3 in the example). You thus need to also override those two subprograms to return correct values.

18.8 Creating your own SQL types

GNATColl comes with a number of predefined types that you can use in your queries. '`gnatcoll_db2ada`' will generate a file using any of these predefined types, based on what is defined in your actual database.

But sometimes, it is convenient to define your own SQL types to better represent the logic of your application. For instance, you might want to define a type that would be for a `Character` field, rather than use the general `SQL_Field_Text`, just so that you can write statements like:

```
declare
C : Character := 'A';
Q : SQL_Query;
begin
Q := SQL_Select (.., Where => Table.Field = C);
end
```

This is fortunately easily achieved by instantiating one generic package, as such

```
with GNATCOLL.SQL_Impl; use GNATCOLL.SQL_Impl;

function To_SQL (C : Character) return String is
begin
return "'" & C & "'";
end To_SQL;

package Character_Fields is new Field_Types (Character, To_SQL);
type SQL_Field_Character is new Character_Fields.Field
with null record;
```

This automatically makes available both the field type (which you can use in your database description, as 'gnatcoll_db2ada' would do, but also all comparison operators like <, >, =, and so on, both to compare with another character field, or with Character Ada variable. Likewise, this makes available the assignment operator = so that you can create INSERT statements in the database.

Finally, the package Character_Fields contain other generic packages which you can instantiate to bind SQL operators and functions that are either predefined in SQL and have no equivalent in GNATColl yet, or that are functions that you have created yourself on your DBMS server.

See the specs of GNATCOLL.SQL_Impl for more details. This package is only really useful when writing your own types, since otherwise you just have to use GNATCOLL.SQL to write the actual queries.

18.9 Query logs

In Chapter 4 [Logging information], page 31 we discovered the logging module of GNATColl. The database interface uses this module to log the queries that are sent to the server.

If you activate traces in your application, the user can then activate one of the following trace handles to get more information on the exchange that exists between the database and the application. As we saw before, the output of these traces can be sent to the standard output, a file, the system logs,...

The following handles are provided:

- SQL.ERROR This stream is activated by default. Any error returned by the database (connection issues, failed transactions,...) will be logged on this stream
- SQL This stream logs all queries that are not SELECT queries, ie mostly all queries that actually modify the database
- SQL.SELECT This stream logs all select queries. It is separated from SQL because very often you will be mostly interested in the queries that impact the database, and logging all selects can generate a lot of output.

18.10 Tasks and databases

As we saw before, the database interface can be used in multi-tasking applications. In such a case, it is recommended that each thread has its own connection to the database, since that is more efficient and you do not have to handle locking.

However, this assumes that the database server itself is thread safe, which most often is the case, but not for `sqlite` for instance. In such a case, you can only connect one per application to the database, and you will have to manage a queue of queries somehow.

18.11 Creating and inspecting databases

The API described earlier in this chapter allows you to query and modify and existing database. However, you first need to create that database.

In most cases, this creation needs to be done by a system administrator with the appropriate rights, and thus will be done as part of the deployement of your application, not the application itself. This is particularly true for client-server databases like `PostgreSQL`.

But in some simpler cases where the database is only manipulated by your application, and potentially only needs to exist while your application is running (often the case for `sqlite`), your application could be responsible for creating the database.

The package `GNATCOLL.SQL.Inspect` provides a number of subprograms to do so, as well as to query the schema of the database (which tables exist, what are their fields,...)

The database schema is represented by a specific data structure in memory. This structure can be initialized through different means:

- either by parsing a file.

  The format of this file was described above (see Section 18.2 [Database schema monitoring], page 85).

- or by querying an existing database.

  This is not the case we are interested in here, but provides a simple way to get started with GNATColl. In particular, you could query your existing database, then dump the schema into a file compatible with the GNATCOLL format. From then on, you would manipulate that file if you need to perform changes in your schema.

  This technics is not as accurate as using file. For instance, there is no way to guess that a table only exists to implement a many-to-many relationship between two other tables. The file provides a high-level view that describes this.

Once you have the schema in memory, you can perform various actions:

- dump it into a file

  Of course, this is only useful if the schema was loaded from some other means.

- create a database with it

  GNATColl will emit the appropriate SQL commands to create the corresponding tables in the database. For this, you will need a connection to the database (see the examples above). Note that in the case of `sqlite`, the database need not exist first, and it will be automatically created if needed.

This input/output mechanism is implemented through an abstract `Schema_IO` tagged type, with various concrete implementations (either `File_Schema_IO` to read or write from/to a file, or `DB_Schema_IO` to read or write from/to a database).

See the specs for more detail on these subprograms.

Index

S

T

U

W