GNATColl Documentation

Release 21.0.0

AdaCore

May 11, 2022

CONTENTS

CHAPTER

ONE

INTRODUCTION TO THE GNAT COMPONENT COLLECTION

The reusable library known as the GNAT Component Collection (GNATColl) is based on one main principle: general-purpose packages that are part of the GNAT technology should also be available to GNAT user application code. The compiler front end, the GNAT Programming Studio (GPS) Interactive Development Environment, and the GNAT Tracker web-based interface all served as sources for the components.

The GNATColl components complement the predefined Ada and GNAT libraries and deal with a range of common programming issues including string and text processing, memory management, and file handling. Several of the components are especially useful in enterprise applications.

1.1 Bug reports

Please send questions and bug reports to report@adacore.com following the same procedures used to submit reports with the GNAT toolset itself.

CHAPTER

TWO

BUILDING GNATCOLL

In the instructions detailed below, it is assumed that you have unpacked the GNATColl package in a temporary directory and that *installdir* is the directory in which you would like to install the selected components.

It is further assumed that you have recent functional GNAT compiler, as well as gprbuild.

2.1 Configuring the build environment

The first step is to configure the build environment. This is done by running the *make setup* command in the root directory of the GNATColl tree. This step is optional if you are satisfied with default values.

On Windows, this requires a properly setup Unix-like environment, to provide Unix-like tools.

The following variables can be used to configure the build process:

General:

prefix Location of the installation, the default is the running GNAT installation root.

INTEGRATED Treat prefix as compiler installation: yes or no (default). This is so that installed gnatcoll project can later be referenced as a predefined project of this compiler; this adds a normalized target subdir to prefix.

BUILD Controls the build options : PROD (default) or DEBUG

PROCESSORS Parallel compilation (default is 0, which uses all available cores)

TARGET For cross-compilation, auto-detected for native platforms

SOURCE_DIR For out-of-tree build

ENABLE_SHARED Controls whether shared and static-pic library variants should be built: yes (default) or no. If you only intend to use static libraries, specify 'no'.

Module-specific:

GNATCOLL_MMAP Whether MMAP is supported: yes (default) or no; this has no effect on Windows where embedded MMAP implementation is always provided.

GNATCOLL_MADVISE Whether MADVISE: yes (default) or no; this has no effect on Windows where MADVISE functionality is unavailable

2.2 Building GNATColl

GNATCOLL Core Module can be built using a GPR project file, to build it is as simple as:

> $ gprbuild gnatcoll.gpr

Though, to build all versions of the library (static, relocatable and static-pic) it is simpler to use the provided Makefile:

> $ make

2.3 Installing GNATColl

Installing the library is done with the following command:

```
make install
```

Note that this command does not try to recompile GNATColl, so you must build it first. This command will install all library variants that were built.

Your application can now use the GNATColl code through a project file, by adding a `with` clause to `gnatcoll.gpr`.

If you wish to install in a different location than was specified at configure time, you can override the "prefix" variable from the command line, for instance:

```
make prefix=/alternate/directory install
```

This does not require any recompilation.

CHAPTER

THREE

SCRIPTS: 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.

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.

The Core component provides support for the following language:

Shell This is a very simple-minded scripting language, which doesn't provide flow-control instructions (*The Shell language*).

Optional components add support for other languages, e.g. Python. Please refer to the corresponding component's documentation.

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" field
Shell> @id %1 5          # Set its "id" field
```

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

Function load (file) 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

Function echo (arg...) 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.

Function clear_cache 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 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.

Class Console *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.

Console.write(text) 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*.

Console.clear() Clears the contents of the console.

Console.flush() Does nothing currently, but is needed for compatibility with python. Output through *Console* instances is not buffered anyway.

Console.isatty(): Boolean Whether the console is a pseudo-terminal. This is always wrong in the case of GNATColl.

Console.read([size]): string Reads at most *size* bytes from the console, and returns the resulting string.

Console.readline([size]): string 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 `<prefix>/share/examples/gnatcoll` 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 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.

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:

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

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

```ada
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 => ...);
```

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");
```

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

Procedure Register_Python_Scripting (Repo, Module_Name) 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*

Exporting standard classes

To be fully functional, GNATColl requires some predefined classes to be exported to all languages (*Classes exported to all languages*). 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.

Procedure Register_Standard_Classes(Repo,Console_Class) 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. 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.

Virtual_Console.Insert_Text (Txt)

Virtual_Console.Insert_Log (Txt)

Virtual_Console.Insert_Error (Txt) 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.

Virtual_Console.Insert_Prompt (Txt) 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

Virtual_Console.Set_As_Default_Console (Script) 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

Virtual_Console.Read (Size, Whole_Line) : String Read either several characters or whole lines from the console. This is called when the user scripts read from their stdin.

Virtual_Console.Set_Data_Primitive (Instance)

Virtual_Console.Get_Instance : 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.

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:

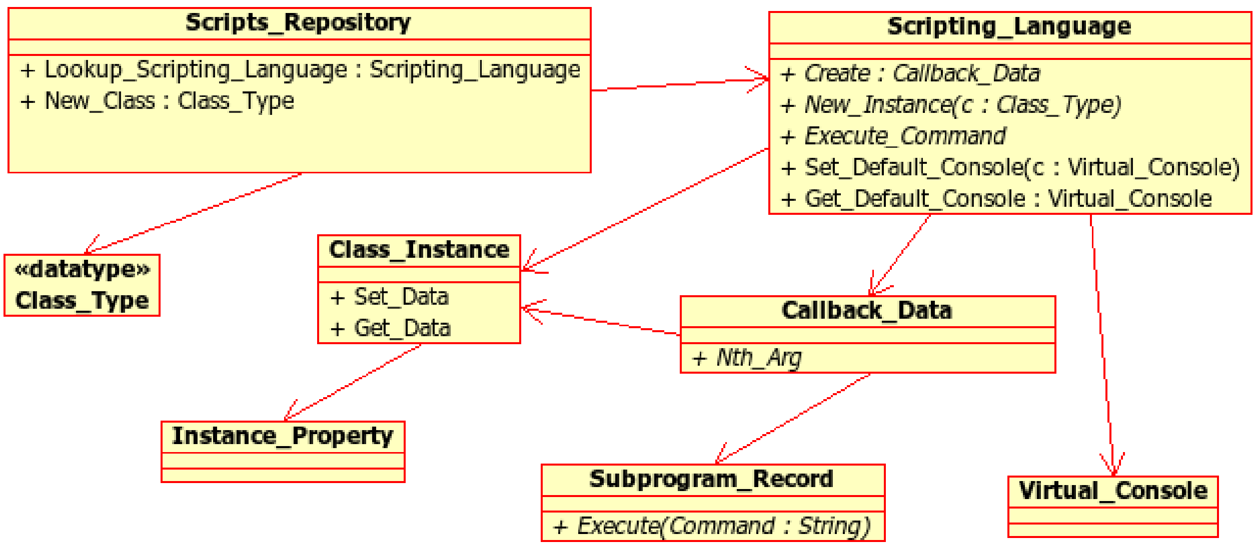
```
declare
   Inst : constant Class_Instance := Nth_Arg (Data, 1);
begin
   C := new My_Console_Record;  --  or your own type
   GNATCOLL.Scripts.Set_Data (Inst, C);
end
```

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.

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:



Class Scripts_Repository 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).

Class Scripting_Language 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.

Class Callback_Data 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),. . .

Record Class_Type 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 Class_Instance 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"
```

Class Instance_Property 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.

Class Subprogram_Record 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.

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) 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);
```

This is 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
```

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"
```

(continues on next page)

(continued from previous page)

```
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.

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!

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.

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
  ```

  (continues on next page)

(continued from previous page)

```
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 = No_Class_Instance then
        Inst := New_Instance (Get_Script (Data), MyClass);
        Set (Tmp.Inst.all, Get_Script (Data), Inst);
        Set_Data (Inst, Tmp);
     end if;
     Set_Return_Value (Data, 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 Multithreading applications and scripts

Python itself is not thread-safe. So a single thread can call the python C API at a time. To enforce this, the python interpreter provides a global interpreter lock, which you must acquire before calling the C API, and release when you are done. To simulate multitasking, the python interpreter will in fact release and reacquire the lock every 100 micro-instructions (opcodes in the python virtual machine), to give a chance to run to other tasks. So this is preemptive multitasking.

The threads that are created in Ada that do not need access to python do not need any special handling. However, those that need access to python must make a special function call before they first call the python C API, so that python can create a thread-specific data for them.

GNATCOLL.Scripts.Python contains a number of subprograms to interact with the global interpreter lock of the python engine. The initialization of your application needs to do two extra calls:

```
Register_Python_Scripting (...);
Initialize_Threads_Support;   --  Also acquires the lock
Begin_Allow_Threads;          --  Releases the lock
```

Whenever a task needs to execute python commands (or basically use any subprogram from *GNATCOLL.Scripts*, it needs to do the following:

```
Ensure_Thread_State;   --  Block all python threads
...  access to python C API as usual
Begin_Allow_Threads;   --  Let other python threads run
```

In some cases, the simplest is to get the lock at the beginning of the task, and release it when done. This assumes the task executes fast enough. In other cases, you will need finer grain control over the lock.

3.2.6 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 (*Traces: Logging information*) 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.

CHAPTER

FOUR

TRACES: 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 (*Defining custom stream types*). 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. In the file name, *$$* is automatically replaced by the process number. *$D* is automatically replaced by the current date. *$T* is automatically replaced by the current date and time. Other patterns of the form *$name*, *${name}*, or *$(name)* are substituted with the value of the named environment variable, if it exists. If ">>" is used instead of ">" to redirect to that stream, the file is appended to, instead of truncated.

"&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" *Logging to syslog*.

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.

  There is are two special cases for the names on this line: they can start with either *"*."* or *".*"* to indicate the settings apply to a whole set of handles. See the example below.

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
```

(continues on next page)

(continued from previous page)

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

--  Applies to FIRST.EXCEPTIONS, LAST.EXCEPTIONS,...
--  and forces them to be displayed on stdout
*.EXCEPTIONS=yes > stdout

--  Applies to MODULE1, MODULE1.FIRST,... This can be used to
--  disable a whole hierarchy of modules.
--  As always, the latest config overrides earlier ones, so the
--  module MODULE1.EXCEPTIONS would be disabled as well.

MODULE1.*=no
```

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.

The function `Parse_Config_File` must be called to indicate that you want to activate the traces. It must also end up finding a configuration file. If it does not, then none of the other functions will ever output anything. This is to make sure your application does not start printing extra output just because you happen to use an external library that uses `GNATCOLL.Traces`. It also ensures that your application will not try to write to `stdout` unless you think it is appropriate (since `stdout` might not even exist in fact).

You then need to declare each of the *trace_handle* (or *logger*) 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.

Function Trace_Handle Create Name Default Stream Factory Finalize 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* (*Log decorators*).

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;
```

(continues on next page)

(continued from previous page)

```
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");
```

An additional subprogram can be used to test for assertions (pre-conditions 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.

4.2.1 Logging unexpected exceptions

A special version of *Trace* is provided, which takes an *Exception_Occurrence* as argument, and prints its message and backtrace into the corresponding log stream.

This procedure will in general be used for unexcepted exceptions. Since such exceptions should be handled by developers, it is possible to configure *GNATCOLL.TRACES* to use special streams for those.

Trace (Me, E) will therefore not used *Me* itself as the log handle, but will create (on the fly, the first time) a new handle with the same base name and and *.EXCEPTIONS* suffix. Therefore, you could put the following in your configuration file:

```
# Redirect all exceptions to stdout
*.EXCEPTIONS=yes >& stdout
```

and then the following code will output the exception trace to stdout:

```
procedure Proc is
   Me : Create ("MYMODULE");
begin
   ...
exception
   when E : others =>
      Trace (Me, E, Msg => "unexcepted exception:");
end Proc;
```

4.2.2 Checking 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;
```

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.MICRO_TIME If active, the time displayed by DEBUG.ABSOLUTE_TIME will use a microseconds precision, instead of milliseconds.

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.MEMORY Every time a message is output, display the amount of memory currently in use by the application.

DEBUG.SPLIT_LINES When this is enabled, messages are split at each newline character. Each line then starts with the name of the logger, indentation level and so on. This might result in more readable output, but is slightly slower.

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 (*Dynamically disabling features*), 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 stream types

We noted above that several predefined types of 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.

You do not need the code below if you simply want to have a new stream in your application (for instance using one for logging Info messages, one for Error messages, and so on). In this case, the function *Create* is all you need.

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 primitive operation *Put*.

The whole output message is given as a single parameter to *Put*:

```
type My_Stream is new Trace_Stream_Record with record
   File : access File_Type;
end record;

procedure Put
  (Stream : in out My_Stream; Str : Msg_Strings.XString)
is
   S : Msg_Strings.Unconstrained_String_Access;
   L : Natural;
begin
   Str.Get_String (S, L);
   Put (Stream.File.all, String (S (1 .. L)));
end Put;
```

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.

A factory is also a tagged object (so that you can store custom information in it), with a single primitive operation, *New_Stream*, in charge of creating and initializing a new stream. This operation 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:

```
type My_Stream_Factory is new Stream_Factory with null record;

overriding function New_Stream
   (Self : My_Stream_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;

Fact : access My_Stream_Factory := new My_Stream_Factory;
Register_Stream_Factory ("mystream", Fact);
```

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 relevant 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.

CHAPTER

FIVE

STRINGS: HIGH-PERFORMANCE STRINGS

The generic package `GNATCOLL.Strings_Impl` (and its default instantiation in `GNATCOLL.Strings`) provides a high-performance strings implementation.

It comes in addition to Ada's own *String* and *Unbounded_String* types, although it attempts to find a middle ground in between (flexibility vs performance).

GNATCOLL.Strings therefore provides strings (named *XString*, as in extended-strings) that can grow as needed (up to *Natural'Last*, like standard strings), yet are faster than unbounded strings. They also come with an extended API, which includes all primitive operations from unbounded strings, in addition to some subprograms inspired from GNATCOLL.Utils and the python and C++ programming languages.

5.1 Small string optimization

GNATCOLL.Strings uses a number of tricks to improve on the efficiency. The most important one is to limit the number of memory allocations. For this, we use a trick similar to what all C++ implementations do nowadays, namely the small string optimization.

The idea is that when a string is short, we can avoid all memory allocations altogether, while still keeping the string type itself small. We therefore use an Unchecked_Union, where a string can be viewed in two ways:

```
Small string

  [f][s][ characters of the string 23 bytes               ]
     f  = 1 bit for a flag, set to 0 for a small string
     s  = 7 bits for the size of the string (i.e. number of significant
          characters in the array)

Big string

  [f][c      ][size     ][data      ][first     ][pad    ]
     f = 1 bit for a flag, set to 1 for a big string
     c = 31 bits for half the capacity. This is the size of the buffer
         pointed to by data, and which contains the actual characters of
         the string.
     size = 32 bits for the size of the string, i.e. the number of
         significant characters in the buffer.
     data = a pointer (32 or 64 bits depending on architecture)
     first = 32 bits, see the handling of substrings below
     pad = 32 bits on a 64 bits system, 0 otherwise.
         This is because of alignment issues.
```

So in the same amount of memory (24 bytes), we can either store a small string of 23 characters or less with no memory allocations, or a big string that requires allocation. In a typical application, most strings are smaller than 23 bytes, so we are saving very significant time here.

This representation has to work on both 32 bits systems and 64 bits systems, so we have careful representation clauses to take this into account. It also needs to work on both big-endian and little-endian systems. Thanks to Ada's representation clauses, this one in fact relatively easy to achieve (well, okay, after trying a few different approaches to emulate what's done in C++, and that did not work elegantly). In fact, emulating via bit-shift operations ended up with code that was less efficient than letting the compiler do it automatically because of our representation clauses.

5.2 Character types

Applications should be able to handle the whole set of Unicode characters. In Ada, these are represented as the Wide_Character type, rather than Character, and stored on 2 bytes rather than 1. Of course, for a lot of applications it would be wasting memory to always store 2 bytes per character, so we want to give flexibility to users here.

So the package GNATCOLL.Strings_Impl is a generic. It has several formal parameters, among which:

- Character_Type is the type used to represent each character. Typically, it will be Character, Wide_Character, or even possibly Wide_Wide_Character. It could really be any scalar type, so for instance we could use this package to represent DNA with its 4-valued nucleobases.
- Character_String is an array of these characters, as would be represented in Ada. It will typically be a String or a Wide_String. This type is used to make this package work with the rest of the Ada world.

Note about Unicode: we could also always use a Character, and use UTF-8 encoding internally. But this makes all operations (from taking the length to moving the next character) slower, and more fragile. We must make sure not to cut a string in the middle of a multi-byte sequence. Instead, we manipulate a string of code points (in terms of Unicode). A similar choice is made in Ada (String vs Wide_String), Python and C++.

5.3 Configuring the size of small strings

The above is what is done for most C++ implementations nowadays. The maximum 23 characters we mentioned for a small string depends in fact on several criteria, which impact the actual maximum size of a small string:

- on 32 bits system, the size of the big string is 16 bytes, so the maximum size of a small string is 15 bytes.
- on 64 bits system, the size of the big string is 24 bytes, so the maximum size of a small string is 23 bytes.
- If using a Character as the character type, the above are the actual number of characters in the string. But if you are using a Wide_Character, this is double the maximum length of the string, so a small string is either 7 characters or 11 characters long.

This is often a reasonable number, and given that applications mostly use small strings, we are already saving a lot of allocations. However, in some cases we know that the typical length of strings in a particular context is different. For instance, GNATCOLL.Traces builds messages to output in the log file. Such messages will typically be at most 100 characters, although they can of course be much larger sometimes.

We have added one more formal parameter to GNATCOLL.Strings_Impl to control the maximum size of small strings. If for instance we decide that a "small" string is anywhere from 1 to 100 characters long (i.e. we do not want to allocate memory for those strings), it can be done via this parameter.

Of course, in such cases the size of the string itself becomes much larger. In this example it would be 101 bytes long, rather than the 24 bytes. Although we are saving on memory allocations, we are also spending more time copying data when the string is passed around, so you'll need to measure the performance here.

The maximum size for the small string is 127 bytes however, because this size and the 1-bit flag need to fit in 1 bytes in the representation clauses we showed above. We tried to make this more configurable, but this makes things significantly more complex between little-endian and big-endian systems, and having large "small" strings would not make much sense in terms of performance anyway.

Typical C++ implementations do not make this small size configurable.

5.4 Task safety

Just like unbounded strings, the strings in this package are not thread safe. This means that you cannot access the same string (read or write) from two different threads without somehow protecting the access via a protected type, locks,…

In practice, sharing strings would rarely be done, so if the package itself was doing its own locking we would end up with very bad performance in all cases, for a few cases where it might prove useful.

As we'll discuss below, it is possible to use two different strings that actually share the same internal buffer, from two different threads. Since this is an implementation detail, this package takes care of guaranteeing the integrity of the shared data in such a case.

5.5 Copy on write

There is one more formal parameter, to configure whether this package should use copy-on-write or not. When copy on write is enabled, you can have multiple strings that internally share the same buffer of characters. This means that assigning a string to another one becomes a reasonably fast operation (copy a pointer and increment a refcount). Whenever the string is modified, a copy of the buffer is done so that other copies of the same string are not impacted.

But in fact, there is one drawback with this scheme: we need reference counting to know when we can free the shared data, or when we need to make a copy of it. This reference counting must be thread safe, since users might be using two different strings from two different threads, but they share data internally.

Thus the reference counting is done via atomic operations, which have some impact on performance. Since multiple threads try to access the same memory addresses, this is also a source of contention in multi-threaded applications.

For this reason, the current C++ standard prevents the use of copy-on-write for strings.

In our case, we chose to make this configurable in the generic, so that users can decide whether to pay the cost of the atomic operations, but save on the number of memory allocations and copy of the characters. Sometimes it is better to share the data, sometimes to systematically copy it. Again, actual measurements of the performance are needed for your specific application.

5.6 Growth strategy

When the current size of the string becomes bigger than the available allocated memory (for instance because you are appending characters), this package needs to reallocate memory. There are plenty of strategies here, from allocating only the exact amount of memory needed (which saves on memory usage, but is very bad in terms of performance), to doubling the current size of the string until we have enough space, as currently done in the GNAT unbounded strings implementation.

The latter approach would therefore allocate space for two characters, then for 4, then 8 and so on.

This package has a slightly different strategy. Remember that we only start allocating memory past the size of small strings, so we will for instance first allocate 24 bytes. When more memory is needed, we multiply this size by 1.5, which some researchers have found to be a good comprise between waste of memory and number of allocations. For very large strings, we always allocate multiples of the memory page size (4096 bytes), since this is what the system will make available anyway. So we will basically allocate the following: 24, 36, 54, 82, 122,…

An additional constraint is that we only ever allocate even number of bytes. This is called the capacity of the string. In the layout of the big string, as shown above, we store half that capacity, which saves one bit that we use for the flag.

5.7 Substrings

One other optimization performed by this package (which is not done for unbounded strings or various C++ implementations) is to optimize substrings when also using copy-on-write.

We simply store the index of the first character of the string within the shared buffer, instead of always starting at the first.

From the user's point of view, this is an implementation detail. Strings are always indexed from 1, and internally we convert to an actual position in the buffer. This means that if we need to reallocate the buffer, for instance when the string is modified, we transparently change the index of the first character, but the indexes the user was using are still valid.

This results in very significant savings, as shown below in the timings for Trim for instance. Also, we can do an operation like splitting a string very efficiently.

For instance, the following code doesn't allocate any memory, beside setting the initial value of the string. It parses a file containing some "key=value" lines, with optional spaces, and possibly empty lines:

```
declare
   S, Key, Value : XString;
   L             : XString_Array (1 .. 2);
   Last          : Natural;
begin
   S.Set (".......");

   --  Get each line
   for Line in S.Split (ASCII.LF) loop

      --  Split into at most two substrings
      Line.Split ('=', Into => L, Last => Last);

      if Last = 2 then
         Key := L (1);
         Key.Trim;    --  Removing leading and trailing spaces

         Value := L (2);
         Value.Trim;

      end if;
   end loop;
end;
```

5.8 API

This package provides a very extensive set of API that apply to *XString*, please check the spec in `gnatcoll-strings_impl.ads` for a fully documented list.

CHAPTER SIX

MEMORY: MONITORING MEMORY USAGE

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.

Like this:

```
procedure Ada
```

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;
```

- 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 *GNAT-*

COLL.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.

Debugging dangling pointer Using a dangling pointer can lead (and usually it does) to no crash or no side effects. Frequently, freed buffers still contains valid data and are still part of pages owned by your process. Probably, this occurs more often on linux compare to windows.

Writing 0 or 0xDD pattern when a memory is freed will be (because of the exception that will be thrown) detected at the first usage of a freed buffer. The crash occurrence will be higher and less random. This makes solid reproducer more easy to build.

For dangling pointer usage debugging, use Memory_Free_Pattern parameter when calling *GNATCOLL.Memory.Configure* procedure.

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

CHAPTER

SEVEN

MMAP: 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;
   Reg  : Mapped_Region;
   Str  : Long.Str_Access;
begin
   File := Open_Read ("/tmp/file_on_disk");
```

(continues on next page)

(continued from previous page)

```
   Reg := Read (File);  *--  map the whole file*
   Close (File);

   Str := Long.Data (File);
   for S in 1 .. Long.Last (File) loop
       Put (Str (S));
   end loop;
   Free (Reg);
end;
```

The above example works for files larger than 2Gb, on 64 bits system (up to a petabyte in fact), on systems that support the *mmap* system call.

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;
   Reg    : Mapped_Region;
   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, Reg, 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;
   Free (Reg);
   Close (File);
end;
```

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*.

CHAPTER

EIGHT

BOYER-MOORE: 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));
```

CHAPTER

NINE

PARAGRAPH FILLING: FORMATTING TEXT

The package *GNATCOLL.Paragraph_Filling* provides several algorithms for filling paragraphs—formatting them to take up the minimal number of lines and to look better. *Knuth_Fill* is based on an algorithm invented by Donald Knuth, and used in TeX. *Pretty_Fill* uses a different algorithm, which was judged by some to produce more aesthetically pleasing output.

More detailed documentation may be found in the comments in the package spec.

CHAPTER

TEN

TEMPLATES: GENERATING TEXT

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

CHAPTER

ELEVEN

EMAIL: PROCESSING EMAIL MESSAGES

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) which contains the necessary subprograms to connect with an SMTP server.

11.1 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 *Parsing messages*.

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.

11.2 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.

11.3 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.

11.4 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.

CHAPTER

TWELVE

RAVENSCAR: PATTERNS FOR MULTITASKING

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.

12.1 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.

12.2 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.

12.3 Timers

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

CHAPTER THIRTEEN

STORAGE POOLS: CONTROLLING MEMORY MANAGEMENT

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.

`GNATCOLL.Storage_Pools.Headers` This pool allows you to allocate memory for the element and reserve extra space before it for a header. This header can be used to store per-element information, like for instance a reference counter, or next and previous links to other elements in the same collection.

  In many cases, this can be used to reduce the number of allocations, and thus speed up the overall application.

CHAPTER

FOURTEEN

VFS: 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.

14.1 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 (*Remote filesystems*).

14.1.1 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*

14.2 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.

14.2.1 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;
```

14.2.2 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.

14.3 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.

CHAPTER

FIFTEEN

TRIBOOLEANS: 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.

CHAPTER

SIXTEEN

GEOMETRY: PRIMITIVE GEOMETRIC OPERATIONS

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;
```

CHAPTER SEVENTEEN

PROJECTS: MANIPULATING GPR FILES

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 Ada 2012 notation, it is fully compatible with Ada 95.

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;

procedure Test_Project is
   Tree  : Project_Tree;
   Files : File_Array_Access;
begin
   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;

   Tree.Unload;
end Test_Project;
```

17.1 Defining a project with user-defined packages and reading them

If you want to use *GNATCOLL.Projects* with a GPR file that contains specific packages and attributes, you must procede in several steps. The following example will show you how to do it:

```
with Ada.Text_IO; use Ada.Text_IO;

with GNAT.Strings;

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

```

(continues on next page)

(continued from previous page)

```
procedure Test_Project is
   Tree         : Project_Tree;
   Project_File : constant Virtual_File :=
      GNATCOLL.VFS.Create (+"path_to_project.gpr");
begin
   --  1
   if Register_New_Attribute
        ("String", "Package_Name") /= ""
      or else Register_New_Attribute
        ("List", "Package_Name", Is_List => True) /= ""
      or else Register_New_Attribute
        ("Indexed", "Package_Name", Indexed => True) /= ""
   then
      raise Program_Error;
   end if;

   --  2
   Tree.Load (Root_Project_Path => Project_File,
              Packages_To_Check => All_Packs);

   declare
      Root : constant Project_Type := Tree.Root_Project;

      --  3
      String_Attribute : constant Attribute_Pkg_String :=
         Build ("Package_Name", "String");
      String_Value : constant String :=
         Root.Attribute_Value (String_Attribute);

      Indexed_Attribute : constant Attribute_Pkg_String :=
         Build ("Package_Name", "Indexed");
      Indexed_Value : constant String :=
         Root.Attribute_Value
           (Indexed_Attribute, Index => "Index");

      List_Attribute : constant Attribute_Pkg_List :=
         Build ("Package_Name", "List");
      List_Value : constant GNAT.Strings.String_List_Access :=
         Root.Attribute_Value (List_Attribute);
   begin
      --  4
      Put_Line ("Package_Name.String: " & String_Value);
      Put_Line ("Package_Name.Indexed (""Index""): "
                & Indexed_Value);
      Put_Line ("Package_Name.List:");
      for Val of List_Value.all loop
         Put_Line ("  Value: " & Val.all);
      end loop;
   end;
end Test_Project;
```

And the corresponding project file:

```
project Path_To_Project is
   package Package_Name is
      for String use "some string"
      for Indexed ("Index") use "other string";
      for List use ("first item", "second item");
   end Package_Name;
end Path_To_Project;
```

Step 1: We register all the attributes that we want for a given package. If the package does not already exists it is created.

Step 2: We load the project into the projects hierarchy. We tell `Tree.Load` to check all packages otherwise it will not load any packages.

Step 3: We read attributes from the project. An attribute can be an `Attribute_Pkg_String` (representing a plain string) or an `Attribute_Pkg_List` (representing a list of strings).

Step 4: We can do something with those values. Here we print the plain string and the content of the list.

This program should output:

```
Package_Name.String: some string
Package_Name.Indexed ("Index"): hello world
Package_Name.List:
  Value: hello
  Value: world
```

CHAPTER

EIGHTEEN

REFCOUNT: 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
```

(continues on next page)

(continued from previous page)

```
   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 are, however, common design patterns 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 lives. 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;
```

CHAPTER

NINETEEN

CONFIG: PARSING 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);
```

(continues on next page)

(continued from previous page)

```
   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.

CHAPTER

TWENTY

POOLS: CONTROLLING ACCESS TO RESOURCES

The package **GNATCOLL.Pools** provides resource pools.

A pool contains a maximum number of resources, which are created on demand. However, once a resource is no longer needed by the client, it is not freed, but instead it is released to the pool, which will then return it again the next time a client requests a resource.

The typical resource is when the creation of the resources is expensive, for instance a connection to a database or a remote server. The lazy creation then provides a faster startup time (as well as more flexibility, since there is no need to allocate dozens of resources if only one will be needed in the end), and more efficient retrieval through the reuse of resources.

The pool in this package is task safe, and is intended as a global variable (or field of a global variable) somewhere in your application.

The resources are implemented as reference-counted types (through *GNATCOLL.Refcount*). As a result, as soon as the client no longer has a handle on them, they are automatically released to the pool and there is no risk that the client forgets to do so.

GNATCOLL.Pools is a generic package where the formal parameters describe the type of resources, how to create them on demand, what should happen when a resource is released, and finally how to free a resource when the pool itself is freed. See `gnatcoll-pools.ads` for a full and up-to-date description of these parameters.

CHAPTER

TWENTYONE

JSON: HANDLING JSON DATA

JSON is a format often used on the web to communicate between a server and a browser, or between servers. It plays a similar role to XML, but it has a much lighter syntax. On the other hand, it doesn't provide advanced features like validation, which XML provides.

The GNATCOLL.JSON package provides an Ada API to decode JSON data from strings and to encode that data back to strings. It also allows one to create and modify JSON data.

21.1 API overview

The entry point for this API is the JSON_Value data type. JSON values can be any of:

- a null value (JSON_Null_Type): all such JSON values are equivalent;
- a boolean value (JSON_Boolean_Type): either true or false;
- an integer value (JSON_Int_Type), they are encoded as an Ada Long_Long_Integer;
- a floating point value (JSON_Float_Type), they are encoded as an Ada Long_Float;
- an UTF-8 encoded string (JSON_String_Type);
- an array of JSON values (JSON_Array_Type);
- a JSON object (JSON_Object_Type), which is a sequence of fields. Each field has a unique name and maps to a JSON value. Depending on the context, this sequence can be processed as a mapping, because each field name is unique, but iterating on fields is deterministic because it is a sequence underneath.

Parsing JSON is as easy as calling the Read function:

```
Data : JSON_Value := Read ("[1, ""foo"", {""foo"": null}]");
```

Encoding to JSON is not any more complex:

```
JSON_String : String := Write (Data);
```

JSON trees (JSON_Value) are available for both inspection and modification:

```
Float_Number : JSON_Value := Create (Float'(1.0));
--  Mere float number

Object : JSON_Value := Get (Get (Data), 3);
--  JSON object from Data: {"foo": null}

Some_Array : JSON_Value :=
```

(continues on next page)

(continued from previous page)

```
   Create (Float_Number & Object & Create (False));
--  Synthetic JSON array: [1.0, {"foo": null}, False]

--  Modify Data in place
Data.Append (Some_Array);
```

21.2 Examples

Here is a complete program demonstrating the use of this API:

```
with Ada.Text_IO;   use Ada.Text_IO;
with GNATCOLL.JSON; use GNATCOLL.JSON;

procedure JSON_Test is
   --  Create a JSON value from scratch
   My_Obj : JSON_Value := Create_Object;
begin
   My_Obj.Set_Field ("field1", Create (1));
   My_Obj.Set_Field ("name", "theName");

   --  Now serialize it. The call below will display:
   --    {"field1": 1, "name": "thename"}
   Put_Line (My_Obj.Write);
end JSON_Test;
```

The above uses the Ada 2005 "dot notation" to call primitive operations (`.Set_Field`, `.Write`), but naturally the more traditional "prefix notation" is also available:

```
Set_Field (My_Obj, "field1", Create (1));
```

It is also possible to create JSON arrays. These are not tagged types, so the prefix notation has to be used. Here is a further example that sets another field in the object we had before (`My_Obj`):

```
declare
   --  Create a JSON array
   My_Arr : JSON_Array := Empty_Array;
begin
   --  Fill it
   Append (My_Arr, Create (1));
   Append (My_Arr, Create ("aString"));

   --  Create a field in My_Obj to hold this array
   My_Obj.Set_Field ("vals", My_Arr);

   --  This will now display:
   --    {"field1": 1, "name": "thename", "vals": [1, "aString"]}
   Put_Line (My_Obj.Write);
end;
```

Similarly to containers from the standard Ada library (from `Ada.Containers`), `GNATCOLL.JSON` features automatic memory management. This means that there is no need for explicit destructors.

The above is all that is needed for most uses of `GNATCOLL.JSON`. To know more about its API, please refer to the gnatcoll-json.ads source file.

CHAPTER

TWENTYTWO

TERMINAL: CONTROLLING THE CONSOLE

Applications generally provide user feedback either via full-fledge graphical interfaces, or via a simpler, console-based output.

The basic support for console-based output is provided directly via *Ada.Text_IO*. But more advanced features are highly system-dependent, and somewhat tricky to develop.

The package *GNATCOLL.Terminal* provide cross-platform support for manipulating colors in terminals, as well as a few basic cursor manipulation subprograms.

22.1 Colors

Most modern terminals support color output, generally with a limit set of colors. On Unix systems, these colors are set by using escape sequences in the output; on Windows systems, these are manipulated by calling functions on a file handle.

GNATCOLL will automatically try to guess whether its output is sent to a color enabled terminal. In general, this will be true when outputing to standard output or standard error, and false when outputing to files or to pipes. You can override this default value to force either color support or black-and-white support.

Here is an example:

```
with Ada.Text_IO;        use Ada.Text_IO;
with GNATCOLL.Terminal;  use GNATCOLL.Terminal;

procedure Test_Colors is
   Info : Terminal_Info;
begin
   Info.Init_For_Stdout (Auto);

   Info.Set_Color (Standard_Output, Blue, Yellow);
   Put_Line ("A blue on yellow line");

   Info.Set_Color (Standard_Output, Style => Reset_All);
   Put_Line ("Back to standard colors -- much better");
end Test_Colors;
```

22.2 Cursors

It is often useful for an application to display some progress indicator during long operations. *GNATCOLL.Terminal* provides a limit set of subprograms to do so, as in:

```
with Ada.Text_IO;        use Ada.Text_IO;
with GNATCOLL.Terminal;  use GNATCOLL.Terminal;

procedure Test_Colors is
   Info : Terminal_Info;
begin
   Info.Init_For_Stdout (Auto);
   for J in 1 .. 1_000 loop
      if J mod 10 = 0 then
         Put ("Processing file" & J'Img & " with long name");
      else
         Put ("Processing file" & J'Img);
      end if;
      delay 0.1;
      Info.Beginning_Of_Line;
      Info.Clear_To_End_Of_Line;
   end loop;
end Test_Colors;
```

CHAPTER TWENTYTHREE

PROMISES: DEFERRING WORK

This package provides a way to synchronize work between some asynchronous workers (or threads).

Promises are a way to encapsulate a yet unknown value, immediately return to the caller, and work in the background to actually execute the work.

For instance, you could have a function that reads some data from a socket. This takes time, and we do not want to block the application while retrieving the data (and if there is an error retrieving it, we certainly want to properly handle it).

The main thread (for instance a graphical user interface) needs to keep processing events and refresh itself. As soon as the data becomes available from the socket, we should let this main thread know so that it can take further action, like post-processing the data and then displaying it.

A general scheme to do that is to have a callback function that is called whenever the work is finished. Promises build on that simple idea so that you can easily chain multiple callbacks to build more complex actions.

See the extensive documentation in `gnatcoll-promises.ads`

CHAPTER

TWENTYFOUR

INDICES AND TABLES

- genindex

INDEX

V

W