GPR User Guide

Release 27.0w

AdaCore

Jun 10, 2026

CONTENTS:

Version 27.0w
Date: Jun 10, 2026

CHAPTER

ONE

INTRODUCTION

This guide is for engineers who want to build, organize, and manage software projects with the **GNAT Project** (GPR) system. It takes a task-oriented approach: each chapter introduces a concept through working examples and explains the reasoning behind the design choices, so you can adapt the patterns to your own projects.

If you are completely new to GPR, start with *Your First Project* and work through the chapters in order. Later chapters build on earlier ones. Experienced users can jump directly to the topic they need; each chapter is largely self-contained.

For a complete specification of the project file language, attributes, and tool command-line interfaces, refer to the **GPR Reference Manual**.

To integrate GPR support into Ada tools - loading project trees or implementing a custom incremental builder - refer to the **GPR2 Quick Start** and the **GPR2 Library Reference**, which document the LibGPR2 Ada library.

1.1 What is GPR?

A GPR *project file* (`.gpr`) is a declarative description of a software component:

- where its source files are,
- what languages they are written in,
- how they should be compiled and linked,
- what is produced (executable, library, or nothing), and
- how this component relates to other components.

GPR is **multi-language**: Ada, C, C++, and many other languages can coexist in the same project tree, each compiled with its own toolchain and settings.

GPR is **tool-agnostic**: the same project file is used by all GPR tools - GPRbuild to compile, GPRclean to remove build results, GPRinstall to deploy, GPRls to inspect sources, and more. You describe the project once; the tools share that description.

GPR is **hierarchical**: complex systems are modeled as a graph of project files, each responsible for one component. Dependencies are expressed as `with` clauses - the same concept as Ada's context clauses. A build of the root project automatically builds everything it depends on.

1.2 Attributes

Every piece of information in a project file is expressed as an **attribute** - a named property that tools read to decide what to do. Attributes cover everything from source directories (`Source_Dirs`) and compiler switches (`Compiler'Switches`) to library names (`Library_Name`) and installation prefixes (`Install'Prefix`). The tools interpret those values; the project file itself contains no executable logic.

CHAPTER

TWO

YOUR FIRST PROJECT

This chapter walks you through creating a minimal GPR project file, building it with GPRbuild, and understanding what each part of the project file means. By the end you will have a working project structure that you can use as a starting point for your own work.

2.1 A minimal project

Create a directory for your project and add a project file:

```
hello/
├── hello.gpr
├── src/
│   └── hello.adb
├── obj/        <- created automatically on first build
└── bin/        <- created automatically on first build
```

The project file `hello.gpr`:

```
project Hello is
   for Source_Dirs use ("src");
   for Object_Dir  use "obj";
   for Exec_Dir    use "bin";
   for Main        use ("hello.adb");
end Hello;
```

And a minimal source file `src/hello.adb`:

```
with Ada.Text_IO;
procedure Hello is
begin
   Ada.Text_IO.Put_Line ("Hello, world!");
end Hello;
```

To build and run:

```
$ gprbuild -P hello.gpr
$ ./bin/hello
Hello, world!
```

That is the complete workflow. The sections below explain each part of the project file.

2.2 Project declaration

```
project Hello is
   ...
end Hello;
```

A project file begins with `project` *Name* `is` and ends with `end` *Name* `;`. The name must match the file name (case-insensitively): project `Hello` lives in `hello.gpr`.

The `project` keyword alone produces a *standard* project - the default kind, which can hold sources and produce executables or object files. Other kinds (`library`, `abstract`, `aggregate`) are covered in later chapters.

2.3 Source directories

```
for Source_Dirs use ("src");
```

`Source_Dirs` is a list of directories, relative to the project file, where GPR tools search for source files. You can list multiple directories:

```
for Source_Dirs use ("src", "gen/src", "platform/src");
```

Glob patterns are also accepted:

```
for Source_Dirs use ("src/**");   --  recursive
```

If `Source_Dirs` is omitted, the directory containing the project file itself is used.

Tip

Keep source directories distinct across projects in the same project tree. Two projects sharing a directory is a common source of hard-to-diagnose duplicate-unit errors.

2.4 Languages

GPR is multi-language. By default, only Ada sources are recognized. To add other languages, list them explicitly:

```
for Languages use ("Ada", "C");
```

GPRbuild compiles only the languages declared here; sources in other languages are ignored.

To build a project that contains only C sources, set `Languages` to `("C")`:

```
for Languages use ("C");
```

2.5 Object and executable directories

```
for Object_Dir use "obj";
for Exec_Dir   use "bin";
```

`Object_Dir` is where compiler-generated files (object files, ALI files, dependency files) are placed. GPRbuild creates this directory automatically before the first build, as long as the path is relative and located under the project directory.

`Exec_Dir` is where linked executables are placed. It defaults to `Object_Dir` when not set. Giving executables their own directory (`"bin"` is conventional) keeps them separate from intermediate build artifacts, making them easier to find, package, or add to `PATH`.

Both attributes default to the project directory if omitted. Declaring them explicitly is strongly recommended so that build products are kept separate from source files.

2.6 Main subprograms

```
for Main use ("hello.adb");
```

`Main` lists the source files that contain program entry points: Ada main subprograms or C `main()` functions. For Ada, only the transitive closure of units required by the entry point is compiled; for other languages, all sources in the project tree are compiled.

See *Building Executables* for multiple entry points, executable naming, and linker switches.

2.7 Compiler switches

To specify compiler switches, declare a `Compiler` package:

```
package Compiler is
   for Switches ("Ada") use ("-gnat2022", "-O1", "-gnatwa");
   for Switches ("C")   use ("-O2", "-Wall");
end Compiler;
```

The index selects which sources the switches apply to: a source file name, a language name, or `others` as a catch-all. See *Building Executables* for the full lookup rules and per-file switch overrides.

Tip

During development, `-gnatwa` (all Ada warnings) and `-gnatVa` (all Ada validity checks) catch many bugs early. Use scenario variables (see *Scenarios*) to select different switch sets for debug and release builds.

2.8 What GPRbuild does

When you run `gprbuild -P hello.gpr`, GPRbuild:

1. Loads `hello.gpr` and any projects it depends on (none here).
2. Determines which sources need (re-)compilation by comparing a recorded build signature against the current situation: source content, compiler switches, and the source's dependencies.
3. Compiles each out-of-date source in parallel into `Object_Dir`.
4. Links an executable for each `Main` source into `Exec_Dir`.

Subsequent builds recompile only what has changed, so incremental builds are fast even in large projects.

2.9 Next steps

- *Managing Sources* - source directory patterns, exclusions, and language-specific naming conventions.
- *Organizing Multi-Project Systems* - splitting a system into multiple project files connected by `with` clauses.
- *Scenarios* - `type` declarations and `case` statements for debug/release builds and platform variants.
- *Libraries* - the `library` project kind.

CHAPTER

THREE

MANAGING SOURCES

This chapter covers how GPR locates source files: directory scanning, explicit file lists, exclusions, and naming conventions. Understanding these mechanisms lets you organize your source tree freely without forcing a particular directory layout.

3.1 How source discovery works

When a project is loaded, GPR tools scan every directory listed in `Source_Dirs` and collects all files whose extensions match the active languages. For Ada, the default extensions are `.ads` (specs) and `.adb` (bodies); for C, `.c` and `.h`; and so on.

Files that do not match any known extension for the declared languages are silently ignored.

3.2 Source directories

`Source_Dirs` accepts a list of directory paths, relative to the project file. Glob patterns are expanded at load time:

```
for Source_Dirs use ("src", "generated/src", "platform/" & Platform);
```

The `**` pattern matches any number of directory levels, enabling a recursive scan:

```
for Source_Dirs use ("src/**");
```

This collects sources from `src/` and every subdirectory beneath it, however deeply nested.

To collect sources *only* from the project file's own directory, use a single dot:

```
for Source_Dirs use (".");
```

To declare a project with no sources - one that exists purely to share attributes with other projects - use the `abstract` qualifier:

```
abstract project Support is
   --  no sources; attributes only
end Support;
```

An `abstract` project may not declare `Main`, `Object_Dir`, or `Exec_Dir`; those attributes are meaningful only for projects that produce build artifacts. See *Project Extension* for a typical use of abstract projects.

Tip

Avoid overlapping `Source_Dirs` between two projects in the same tree. If the same source file is visible to two projects, a duplicate-unit error is reported at load time.

3.3 Explicit source lists

For finer control, you can enumerate sources explicitly instead of scanning a directory.

`Source_Files` lists individual file names (not paths) directly in the project file:

```
for Source_Files use ("utils.ads", "utils.adb", "config.ads");
```

Only the named files are included, even if other source files exist in `Source_Dirs`.

For larger lists, `Source_List_File` points to a text file containing one file name per line:

```
for Source_List_File use "sources.txt";
```

This is useful when the list is generated by a build script or version control system.

3.4 Excluding sources

`Excluded_Source_Files` removes specific files from an otherwise directory-scanned project:

```
for Excluded_Source_Files use ("old_impl.adb", "stub.ads");
```

The complement of `Source_List_File`, `Excluded_Source_List_File` points to a text file listing the files to exclude:

```
for Excluded_Source_List_File use "excluded.txt";
```

Exclusions apply after the directory scan, so you can keep sources in the same directory without including all of them in the build.

3.5 Ada naming conventions

The default naming convention for Ada follows the GNAT standard:

- Specs use the `.ads` extension, bodies use `.adb`.
- Child unit names use `-` as the separator: the body of `My_Lib.Utils` lives in `my_lib-utils.adb`.
- Unit names map to file names in lower case.

Ada source files and unit names are tightly coupled: when a file with the right suffix (`.ads` or `.adb`) is found in a source directory, GPR derives the corresponding unit name by reversing the naming convention (stripping the suffix, replacing the dot-replacement character back to `.`, and so on). If the result is not a valid Ada identifier, the file is silently ignored.

For example, `my_source__unix.adb` maps to `my_source__unix` - a name containing a double underscore, which is not valid in Ada. GPR will not recognize this file as an Ada source during directory scanning.

Tip

To include a file whose name would otherwise be rejected, add an explicit per-unit mapping in the `Naming` package (see *Per-unit overrides* below).

If your project uses a different convention, the `Naming` package lets you override it.

3.5.1 Changing the dot replacement character

To use `__` instead of `-` as the child-unit separator:

```
package Naming is
   for Dot_Replacement use "__";
end Naming;
```

With this setting, the body of `My_Lib.Utils` is expected in `my_lib__utils.adb`.

3.5.2 Changing the default suffixes

```
package Naming is
   for Spec_Suffix ("Ada") use ".1.ada";
   for Body_Suffix ("Ada") use ".2.ada";
end Naming;
```

3.5.3 Per-unit overrides

Individual units can be mapped to arbitrary file names:

```
package Naming is
   for Spec ("My_Lib.Utils") use "ml_utils_s.ada";
   for Body ("My_Lib.Utils") use "ml_utils_b.ada";
end Naming;
```

Per-unit mappings take precedence over suffix rules. This is the mechanism GPRname uses when it generates a project file for a source tree with non-standard naming (see *Working with Tools*).

3.6 Multiple suffixes per language

`Body_Suffix` and `Spec_Suffix` accept a single value per language. A project that contains source files with different suffixes for the same language - for example `.cc` and `.cpp` both for C++ - cannot be described by a single suffix declaration.

Two options are available:

- **Rename all files to a common suffix.** This requires no project structure changes and is the simplest solution when the source tree is under your control.
- **Split into two projects with `limited with`.** Create a companion project that covers the extra suffix, sharing the same `Source_Dirs` and `Object_Dir` as the primary project. The companion imports the primary with a regular `with`; the primary references the companion back with `limited with` to break what would otherwise be a circular dependency. The companion also sets `Spec_Suffix` to a value that matches no file, so it does not claim spec files already owned by the primary. The compiler settings are shared by renaming the package:

```
--  prj.gpr  (handles .cpp files)
limited with "prj_cc_support";

project Prj is
   for Languages   use ("C++");
   for Source_Dirs use ("src");
   for Object_Dir  use "obj";

   package Naming is
      for Body_Suffix ("C++") use ".cpp";
      for Spec_Suffix ("C++") use ".h";
   end Naming;

   package Compiler is
      for Switches ("C++") use ("-O2");
   end Compiler;
end Prj;
```

```
--  prj_cc_support.gpr  (handles .cc files)
with "prj";

project Prj_Cc_Support is
   for Languages   use ("C++");
   for Source_Dirs use Prj'Source_Dirs;
   for Object_Dir  use Prj'Object_Dir;

   package Naming is
      for Body_Suffix ("C++") use ".cc";
      for Spec_Suffix ("C++") use ".no_match";  --  no .no_match files exist
   end Naming;

   package Compiler renames Prj.Compiler;
end Prj_Cc_Support;
```

Build with `gprbuild -P prj.gpr`: GPRbuild loads both projects and compiles `.cpp` files via `Prj` and `.cc` files via `Prj_Cc_Support`, depositing all objects in the shared `obj/` directory.

CHAPTER

FOUR

BUILDING EXECUTABLES

This chapter covers how GPRbuild compiles sources and links executables: declaring entry points, naming outputs, controlling linker and binder switches, and common build options.

4.1 Entry points

An executable is produced for each source file listed in the `Main` attribute of the root project:

```
project Hello is
   for Source_Dirs use ("src");
   for Object_Dir  use "obj";
   for Exec_Dir    use ".";
   for Main        use ("hello.adb", "gen.adb");
end Hello;
```

`Main` takes simple file names - the base name only, no directory component. GPRbuild searches for each main among the sources of the project tree. It compiles the transitive closure of sources required by each entry point and links one executable per entry point, placed in `Exec_Dir`. Sources not reachable from any entry point are not compiled.

To build only a subset of the declared mains, name them on the command line:

```
$ gprbuild -P hello.gpr hello.adb
```

4.2 Executable names

By default the executable name is derived from the main source file name: `hello.adb` produces `hello` (complemented by platform-specific extension, such as `hello.exe` on Windows).

To override the name permanently, use the `Executable` attribute in the `Builder` package:

```
package Builder is
   for Executable ("hello.adb") use "greet";
   for Executable ("gen.adb")   use "codegen";
end Builder;
```

When building a single main, the output name can also be overridden on the command line with `-o`:

```
$ gprbuild -P hello.gpr hello.adb -o test.exe
```

`-o` is only valid when exactly one main is being built.

4.3 Compiler switches

Compiler switches are declared in the `Compiler` package, indexed by language name or source file name:

```
package Compiler is
   for Switches ("Ada") use ("-gnat2022", "-O1", "-gnatwa");
   for Switches ("C")   use ("-O2", "-Wall");
end Compiler;
```

For each source, GPRbuild looks up `Switches` in this order and uses the **first match only** - switches do not accumulate:

1. The source file name (exact match or glob pattern).
2. The source's language name.
3. `others` - the catch-all.

This makes it straightforward to apply special treatment to individual files while keeping a common baseline for everything else:

```
package Compiler is
   for Switches ("Ada")             use ("-gnat2022", "-gnatwa");
   for Switches ("generated_*.adb") use ("-gnat2022", "-gnatws");  --  no warnings
   for Switches ("big_table.adb")   use ("-gnat2022", "-O3");
   for Switches (others)            use ("-O2");         --  C, C++, ...
end Compiler;
```

Here `generated_*.adb` and `big_table.adb` each get their own switch set; all other Ada sources use the `"Ada"` entry; everything else falls through to `others`.

Tip

During development, `-gnatwa` (all Ada warnings) and `-gnatVa` (all Ada validity checks) catch many bugs early. Use scenario variables (see *Scenarios*) to select different switch sets for debug and release builds.

4.4 Linker switches

Linker switches are declared in the `Linker` package:

```
package Linker is
   for Switches ("Ada") use ("-lm", "-lpthread");
end Linker;
```

The index follows the same lookup order as `Compiler'Switches`: source file name or glob first, then language name, then `others`.

Additional link libraries can also be declared with `Linker_Options` in any source file (Ada pragma or GNAT-specific source annotation); GPRbuild collects these automatically from ALI files.

4.5 Binder switches

For Ada programs, GPRbuild invokes the binder (`gnatbind`) before linking. Binder switches go in the `Binder` package:

```
package Binder is
   for Switches ("Ada") use ("-Es");  --  symbolic traceback
end Binder;
```

4.6 Global build switches

The `Builder` package accepts a `Global_Compilation_Switches` attribute that prepends switches to every compilation in the project tree, regardless of language:

```
package Builder is
   for Global_Compilation_Switches ("Ada") use ("-gnatwa");
   for Global_Compilation_Switches ("C")   use ("-Wall");
end Builder;
```

Unlike `Compiler'Switches`, these are applied globally - including to imported projects - and cannot be overridden per source file. Use them sparingly; prefer `Compiler'Switches` for project-local settings.

4.7 Common GPRbuild options

-X*name=value*
: Set the value of an external variable. This is the primary way to select build configurations such as debug or release. See *Scenarios* for how external variables are declared and used in project files.

-j*N*
: Use *N* parallel compilation jobs. `-j0` uses all available cores (the default). `-j1` forces sequential builds, which is useful for reading diagnostics in order.

-f
: Force recompilation of all sources regardless of their build signatures.

-c
: Compile only; do not bind or link.

-v
: Verbose output: print each compilation and link command as it is executed.

-q
: Quiet output: suppress informational messages; show only warnings and errors.

-k
: Keep going after a compilation error: compile as many sources as possible before stopping.

4.8 Out-of-tree builds

By default, build artifacts are written to the directories declared in the project file. To redirect all artifacts to a separate directory without modifying the project file, use `--relocate-build-tree`:

```
$ gprbuild -P my_proj.gpr --relocate-build-tree=/tmp/build
```

Each artifact directory is mirrored under the given path, keeping the source tree untouched. See the *GPR Reference Manual*, chapter *Out-of-Tree Builds*, for the full description including the `--root-dir` option for project trees that span multiple directories.

CHAPTER

FIVE

SCENARIOS

A *scenario* is a named configuration of a project - for example, a debug build, a release build, or a platform-specific variant. GPR expresses scenarios through *typed string variables* whose values are supplied from outside the project file, allowing a single project to describe multiple build configurations without duplication.

5.1 External variables

An external variable is a string value passed to GPR tools via the `-X` command-line switch:

```
$ gprbuild -P my_proj.gpr -XBUILD=release
```

Inside the project file, the value is read with the `external` function:

```
Build : Build_Type := external ("BUILD", "debug");
```

The second argument is the default value used when the variable is not set on the command line.

5.1.1 External_As_List

When an external variable carries a list of values encoded as a single string, `External_As_List` splits it on a given separator and returns a string list:

```
for Source_Dirs use External_As_List ("EXTRA_DIRS", ":");
```

If the environment contains `EXTRA_DIRS=/opt/include:/home/user/src`, the attribute receives `("/opt/include", "/home/user/src")`. This is useful for injecting additional source directories, switches, or other list-valued attributes from the build environment without modifying the project file.

5.2 Typed variables

To constrain the set of accepted values and enable `case` statements, declare a string type first:

```
type Build_Type is ("debug", "release", "release_checks");

Build : Build_Type := external ("BUILD", "debug");
```

If the value passed via `-X` is not in the declared type, a load error is reported. This catches typos early.

5.3 case statements

A `case` statement selects attribute values based on the current scenario variable:

```
project My_App is
   type Build_Type is ("debug", "release");
   Build : Build_Type := external ("BUILD", "debug");

   for Source_Dirs use ("src");
   for Object_Dir  use "obj/" & Build;
   for Exec_Dir    use "bin/" & Build;
   for Main        use ("main.adb");

   package Compiler is
      case Build is
         when "debug" =>
            for Switches ("Ada") use ("-g", "-gnatwa", "-gnatVa");
         when "release" =>
            for Switches ("Ada") use ("-O2", "-gnatn");
      end case;
   end Compiler;
end My_App;
```

A few things to note:

- The `case` expression must be a typed scenario variable.
- Every value of the type must be covered; use `when others` to provide a catch-all.
- `case` statements can appear at the project level and inside packages.
- Variable references (`& Build`) are allowed in attribute values, making it easy to keep debug and release objects in separate directories.

With this project, `gprbuild -P my_proj.gpr` builds a debug binary; `gprbuild -P my_proj.gpr -XBUILD=release` builds an optimized one.

5.3.1 Selecting platform-specific sources

When a `case` statement contains Ada naming exceptions (`for Body` or `for Spec` clauses in the `Naming` package), the source files named in the inactive branches are automatically removed from the project view. This makes it straightforward to keep platform-specific variants in the same source directory without triggering duplicate-unit errors:

```
type OS_Type is ("linux", "windows");
OS : OS_Type := external ("OS", "linux");

package Naming is
   case OS is
      when "linux"   => for Body ("Foo") use "foo_linux.adb";
      when "windows" => for Body ("Foo") use "foo_windows.adb";
   end case;
end Naming;
```

When building with `-XOS=linux`, `foo_windows.adb` is excluded from the project view even though it lives in the source directory.

The same result can be achieved for any language using `Excluded_Source_Files` in a `case` statement:

```
case OS is
   when "linux"   => for Excluded_Source_Files use ("foo_windows.c");
   when "windows" => for Excluded_Source_Files use ("foo_linux.c");
end case;
```

A third approach is to place target-specific sources in dedicated subdirectories. Declare the common directory first, then extend `Source_Dirs` in the `case` statement using the project's own attribute value:

```
src/
├── common/
├── linux/
└── windows/
```

```
for Source_Dirs use ("src/common");

case OS is
   when "linux"   =>
      for Source_Dirs use My_Proj'Source_Dirs & ("src/linux");
   when "windows" =>
      for Source_Dirs use My_Proj'Source_Dirs & ("src/windows");
end case;
```

This layout scales well when many files differ between targets and avoids the need for per-file exclusions or naming exceptions.

5.4 Multiple scenario variables

A project can declare any number of scenario variables, each controlling an independent dimension of the configuration:

```
project My_App is
   type Build_Type is ("debug", "release");
   type OS_Type    is ("linux", "windows", "macos");

   Build : Build_Type := external ("BUILD", "debug");
   OS    : OS_Type    := external ("OS",    "linux");

   for Source_Dirs use ("src/common");
   for Object_Dir  use "obj/" & Build & "-" & OS;
   for Exec_Dir    use "bin/" & Build & "-" & OS;
   for Main        use ("main.adb");

   package Compiler is
      case Build is
         when "debug"   =>
            for Switches ("Ada") use ("-g", "-gnatwa");
         when "release" =>
            for Switches ("Ada") use ("-O2", "-gnatn");
      end case;
   end Compiler;

   case OS is
      when "windows" =>
```

(continues on next page)

(continued from previous page)

```
         for Source_Dirs use My_App'Source_Dirs & ("src/windows");
      when others =>  --  linux, macos
         for Source_Dirs use My_App'Source_Dirs & ("src/posix");
   end case;
   ...
end My_App;
```

Tip

Keep the number of scenario variables small and their names consistent across projects in the same tree. Convention: `BUILD` for debug/release, `TARGET` for cross-compilation target, `RUNTIME` for Ada runtime variant. Consistent naming lets callers pass `-X` switches once at the root and have them apply everywhere.

5.5 Sharing scenario variables

When a project tree has multiple projects, all should use the same typed variable declaration for a given scenario. The common pattern is to declare the type and variable in an `abstract` project and import it everywhere:

```
--  config.gpr
abstract project Config is
   type Build_Type is ("debug", "release");
   Build : Build_Type := external ("BUILD", "debug");
end Config;
```

```
--  my_comp.gpr
with "config.gpr";
project My_Comp is
   package Compiler is
      case Config.Build is
         when "debug"   => for Switches ("Ada") use ("-g");
         when "release" => for Switches ("Ada") use ("-O2");
      end case;
   end Compiler;
   ...
end My_Comp;
```

```
--  my_app.gpr
with "config.gpr";
with "my_comp.gpr";
project My_App is
   package Compiler is
      case Config.Build is
         when "debug"   => for Switches ("Ada") use ("-g", "-gnatwa");
         when "release" => for Switches ("Ada") use ("-O2");
      end case;
   end Compiler;
   ...
end My_App;
```

A single `-XBUILD=release` on the command line then applies consistently to the whole tree.

Warning

External variable names are global to the project tree: two projects that both read the same variable name will always receive the same value. If their typed declarations differ - for example one accepts `("debug", "release")` and another accepts `("debug", "release", "release_checks")` - then passing a value valid for the wider type but not the narrower one causes a load error. To avoid this, share a single type declaration (as shown above) so that all projects in the tree agree on the set of accepted values.

CHAPTER

SIX

LIBRARIES

A *library project* builds a static or shared library from its sources instead of producing an executable. Other projects import the library using a `with` clause and link against it automatically.

6.1 Declaring a library project

Add the `library` qualifier and the two required library attributes:

```
library project My_Lib is
   for Source_Dirs  use ("src");
   for Object_Dir   use "obj";
   for Library_Name use "mylib";
   for Library_Dir  use "lib";
end My_Lib;
```

`Library_Name`
: The base name of the library. The actual file name produced depends on whether the library is an archive or a shared library, and on the target platform - both of which are determined by the active configuration.

`Library_Dir`
: The directory where the library file is placed. Must be distinct from `Object_Dir`.

A library project cannot declare a `Main` attribute; it produces a library, not an executable.

6.2 Library kinds

The `Library_Kind` attribute selects the type of library to build:

```
library project My_Lib is
   ...
   for Library_Kind use "static";   --  default
end My_Lib;
```

`"static"`
: A static archive linked directly into each executable that uses it. This is the default.

`"relocatable"`
: A shared library (`.so` on Linux, `.dll` on Windows) loaded at program start. Only one copy lives in memory when multiple programs use it.

`"static-pic"`
: A position-independent static archive. Needed when the library will itself be bundled into a shared library.

Tip

Use scenario variables (see *Scenarios*) to build both a static and a shared variant from the same project file without duplicating declarations.

6.3 Stand-alone libraries

A *stand-alone library* (SAL) bundles its own Ada elaboration code so that it can be loaded and initialized independently of the main program's elaboration sequence. This is required for plugins and shared libraries loaded at runtime.

`Library_Interface` lists the Ada *unit names* that form the library's public API. `Library_Standalone` controls the degree of self-containment:

`"standard"`
: The library has its own elaboration code for the units listed in `Library_Interface`. Ada dependencies from the rest of the program's elaboration are still expected to be present at load time. This is the default when `Library_Interface` is set.

`"encapsulated"`
: Like `"standard"`, but all Ada library dependencies must themselves be standalone libraries. This produces a fully self-contained library that includes the Ada runtime, with no external Ada elaboration dependencies.

`"no"`
: No standalone elaboration is generated. `Library_Interface` is used only to declare which units form the public API (for visibility purposes), without making the library self-elaborating.

Example - standard SAL:

```
library project My_Lib is
   for Source_Dirs        use ("src");
   for Object_Dir         use "obj";
   for Library_Name       use "mylib";
   for Library_Dir        use "lib";
   for Library_Kind       use "relocatable";
   for Library_Interface  use ("My_Lib", "My_Lib.Utils");
   for Library_Standalone use "standard";
end My_Lib;
```

Example - encapsulated SAL (fully self-contained plugin):

```
library project My_Plugin is
   for Source_Dirs        use ("src");
   for Object_Dir         use "obj";
   for Library_Name       use "myplugin";
   for Library_Dir        use "lib";
   for Library_Kind       use "relocatable";
   for Library_Interface  use ("My_Plugin");
   for Library_Standalone use "encapsulated";
end My_Plugin;
```

Warning

Because an encapsulated library includes the Ada runtime, loading two encapsulated libraries in the same process results in duplicate runtime state, which leads to undefined behavior. A program should load at most one encapsulated library at a time.

6.4 Controlling source visibility

The `Interfaces` attribute restricts which source files of a library project are visible to importing projects. It takes a list of source file names:

```
library project My_Lib is
   for Source_Dirs  use ("src");
   for Object_Dir   use "obj";
   for Library_Name use "mylib";
   for Library_Dir  use "lib";
   for Interfaces   use ("my_lib.ads", "my_lib-utils.ads");
end My_Lib;
```

Units whose source files are not listed in `Interfaces` can still be used internally by the library but cannot be depended on from importing projects. For shared libraries, only the symbols from the interface units are exported; all other symbols are kept internal to the library.

`Interfaces` and `Library_Interface` serve different purposes and can be used together: `Interfaces` controls source-level visibility and symbol export for shared libraries, while `Library_Interface` declares the Ada units forming the public API and drives standalone elaboration behavior via `Library_Standalone`.

6.5 Aggregate library projects

An *aggregate library project* builds a single library by collecting the object files from a set of constituent projects. It combines the `aggregate` and `library` qualifiers and requires `Library_Name`, `Library_Dir`, and `Project_Files`:

```
aggregate library project Full_Lib is
   for Project_Files use ("module_a/a.gpr",
                          "module_b/b.gpr");
   for Library_Name use "full";
   for Library_Dir  use "lib";
end Full_Lib;
```

The resulting library can be of any kind (`static`, `relocatable`, `static-pic`), independently of the library kinds of the constituent projects. Unlike a plain aggregate project, an aggregate library project **can** be `with`-ed by any other project and may appear anywhere in the dependency graph.

Note

An aggregate library project is distinct from a regular aggregate project (see *Aggregate Projects*). In particular, it cannot set external variable values via `External` - only plain aggregate projects support that attribute.

It is advisable to declare `Object_Dir` when constituent sources may need to be recompiled with different flags. GPRbuild stores the recompiled objects there, keeping the constituent projects' own directories untouched.

`Library_Interface`, `Interfaces`, and `Library_Standalone` work as usual, referring to sources from the constituent projects.

CHAPTER

SEVEN

ORGANIZING MULTI-PROJECT SYSTEMS

Real-world software is rarely a single component. This chapter explains how to split a system into multiple project files, express dependencies between them, and control what is visible across project boundaries.

7.1 Why multiple projects?

A single project file works well for a self-contained component. As a system grows, splitting it into multiple projects brings several benefits:

- **Incremental builds**: only the components that changed are recompiled.
- **Reuse**: a library project can be imported by many applications without copying sources.
- **Encapsulation**: each project controls which of its sources are visible to importers (see *Libraries*).
- **Separate build settings**: each project can have its own compiler switches, object directory, and languages.

7.2 Importing a project with `with`

A project declares its dependencies at the top of the file using `with` clauses, one per imported project:

```
with "libs/utils/utils.gpr";
with "libs/crypto/crypto.gpr";

project My_App is
   for Source_Dirs use ("src");
   for Object_Dir  use "obj";
   for Exec_Dir    use "bin";
   for Main        use ("main.adb");
end My_App;
```

The path in a `with` clause is relative to the importing project file. GPRbuild loads all projects in the tree, then builds each source in the order required by the actual source dependency graph.

GPR also searches for project files in the directories listed in the `GPR_PROJECT_PATH` environment variable, so installed libraries can be imported by name without a path:

```
with "gnatcoll.gpr";
```

7.2.1 Accessing attributes and variables of a withed project

After importing a project, you can read its attributes and variables using the *Project*.*Attribute* notation. This is commonly used to reuse settings declared in a shared project:

```
with "common_settings.gpr";

project My_App is
   --  Read a variable defined in Common_Settings
   Build_Mode : Common_Settings.Build_Mode_Type :=
      Common_Settings.Build_Mode;

   --  Reuse the Compiler package verbatim
   package Compiler renames Common_Settings.Compiler;

   --  Append to the imported switches
   package Linker is
      for Switches ("Ada") use
         Common_Settings.Linker'Switches ("Ada") & ("-lm");
   end Linker;
end My_App;
```

The `renames` clause makes the package an alias of the original; any change in `Common_Settings.Compiler` is automatically reflected.

7.3 Limited `with`

A plain `with` gives the importing project full visibility over the withed project's sources, attributes, variables, and packages. Circular plain `with` dependencies are not allowed: if project A imports B and B imports A, a load error is reported.

When two projects need their sources to be mutually visible - for example, two components that are mutually recursive at the Ada unit level - a `limited with` can be used:

```
limited with "component_b.gpr";

project Component_A is
   for Source_Dirs use ("src_a");
   ...
end Component_A;
```

With a `limited with`, the sources of the withed project are visible for compilation, but its attributes, variables, and packages are not accessible from the importing project.

Use `limited with` sparingly. A need for it often indicates that the components involved should be reorganized into a shared lower-level library that both can depend on.

7.4 The project tree

The set of projects reachable from the root project through `with` and `limited with` clauses forms the *project tree*. Loading the tree resolves all source directories, attributes, and naming conventions across all projects. The actual compilation order is then determined by the source dependency graph, independently of the project import structure.

7.5 Typical layout

A common layout for a multi-project system:

```
myproject/
├── myproject.gpr          <- root project (executable)
├── src/
├── obj/
├── libs/
│   ├── utils/
│   │   ├── utils.gpr      <- library project
│   │   ├── src/
│   │   └── obj/
│   └── crypto/
│       ├── crypto.gpr     <- library project
│       ├── src/
│       └── obj/
└── common_settings.gpr    <- abstract project (shared settings)
```

Each library has its own project file, source directory, and object directory. The root project imports all libraries it needs.

CHAPTER

EIGHT

PROJECT EXTENSION

Project extension lets one project inherit sources and settings from another and selectively override them. A common use case is patching a third-party library: the extending project replaces only the files that need to change while reusing everything else unchanged.

8.1 Basic syntax

```
project Patched extends "original/original.gpr" is
   --  Source files here shadow those in original.gpr.
   --  All other sources and settings are inherited.
end Patched;
```

The path in `extends` is relative to the extending project file.

8.2 What is inherited

An extending project inherits:

- **Sources** - all source files from the base are implicitly present. A source file in the extending project whose *base name* matches a base source shadows it.
- **Attribute values** - project-level attributes not declared in the extending project are inherited from the base.
- **Packages** - any package not declared in the extending project is inherited in full from the base.

Exception: `Linker'Linker_Options` is never inherited.

`Object_Dir` and `Library_Dir` must always be overridden in the extending project; the two projects cannot share the same output directories.

```
project Patched extends "original/original.gpr" is
   for Object_Dir  use "obj";
   for Library_Dir use "lib";  -- only for library projects
end Patched;
```

8.3 Overriding sources

Place a source file with the same base name in the extending project's source directory. The extending project's version is used and the base version is ignored entirely.

To remove an inherited source without providing a replacement, list it in `Excluded_Source_Files`:

```
project Patched extends "original/original.gpr" is
   for Object_Dir use "obj";
   for Excluded_Source_Files use ("obsolete.adb");
end Patched;
```

8.4 Overriding attributes and packages

Attribute values are overridden by simply re-declaring them. Package declarations, however, **replace the inherited package entirely** - no per-attribute inheritance occurs within a package:

```
project Patched extends "original/original.gpr" is
   for Object_Dir use "obj";

   --  This replaces Original's entire Compiler package:
   package Compiler is
      for Switches ("Ada") use ("-O2");
   end Compiler;
end Patched;
```

To *amend* a package rather than replace it, use `package X extends Base.X`:

```
project Patched extends "original/original.gpr" is
   for Object_Dir use "obj";

   --  Inherit Original's Compiler switches and add one more:
   package Compiler extends Original.Compiler is
      for Switches ("Ada") use Original.Compiler'Switches ("Ada") & ("-gnatwa");
   end Compiler;
end Patched;
```

8.5 Abstract base projects

A common pattern is to declare common settings in an `abstract` base project and extend it for each concrete build variant. The concrete project still needs to declare its own project-specific attributes (such as `Object_Dir`) for its own purposes:

```
--  base.gpr
abstract project Base is
   type Build_Type is ("debug", "release");
   Build : Build_Type := external ("BUILD", "debug");

   package Compiler is
      case Build is
         when "debug"   => for Switches ("Ada") use ("-g", "-gnatwa");
         when "release" => for Switches ("Ada") use ("-O2");
      end case;
   end Compiler;
end Base;
```

```
--  my_app.gpr
project My_App extends "base.gpr" is
```

(continues on next page)

(continued from previous page)

```
   for Source_Dirs use ("src");
   for Object_Dir  use "obj/" & Base.Build;
   for Exec_Dir    use "bin/" & Base.Build;
   for Main        use ("main.adb");
end My_App;
```

8.6 Import redirection

When a project *D* extends *A*, and *D* also imports *C* which extends *B*, and *A* imports *B* - then within *D*'s project tree, *A*'s import of *B* is automatically redirected to *C*. The build system connects these automatically when both extensions are present in the tree.

For example, suppose *B* defines a variable that *A* references:

```
--  b.gpr
library project B is
   Build_Suffix : constant String := "debug";
   for Source_Dirs  use ("src");
   for Object_Dir   use "obj";
   for Library_Name use "b";
   for Library_Dir  use "lib";
end B;
```

```
--  a.gpr
with "b.gpr";
project A is
   for Source_Dirs use ("src");
   for Object_Dir  use "obj-" & B.Build_Suffix;  --  "obj-debug"
end A;
```

C extends *B* and overrides the variable:

```
--  c.gpr
library project C extends "b.gpr" is
   Build_Suffix : constant String := "release";
   for Object_Dir  use "obj";
   for Library_Dir use "lib";
end C;
```

D extends *A* and imports *C*. When *D* is loaded, *A*'s import of *B* is redirected to *C*, so *A*'s `Object_Dir` becomes `"obj-release"`:

```
--  d.gpr
with "c.gpr";
project D extends "a.gpr" is
   for Object_Dir use "obj";
end D;
```

Since a project tree may not depend on both an extending and an extended project, this mechanism can only be used in simple cases. That is where `extends all` solves the issue for more complex structures.

8.7 extends all

`extends all` implicitly creates an extending project for every project in the import closure of the named base and applies import redirection throughout the entire subtree in one step. This makes it practical to amend any constituent of a large project hierarchy without manually extending every project along the import path.

```
project Full_Override extends all "original/original.gpr" is
   for Object_Dir use "obj";
end Full_Override;
```

The same inheritance rules apply as with a plain `extends`: each constituent inherits its base's sources, attributes, and packages with the same semantics described above. A source file placed in `Full_Override`'s source directory shadows the matching file wherever it originates in the tree.

To replace a specific constituent - for example to substitute instrumented sources for one particular library - import an explicit extending project for that constituent from within the `extends all` project. The explicit extension takes the place of the corresponding implicit one, and import redirection propagates it consistently across the whole subtree:

```
with "instrumented_lib.gpr";  --  extends "libs/my_lib/my_lib.gpr"

project Full_Override extends all "original/original.gpr" is
   for Object_Dir use "obj";
end Full_Override;
```

CHAPTER

NINE

AGGREGATE PROJECTS

This chapter covers *aggregate projects* - a project kind that groups independent project subtrees for a single build invocation. The related *aggregate library project* kind, which produces a library artifact from constituent projects, is covered in *Libraries*.

Note

Despite the similar name and the shared `aggregate` qualifier, aggregate projects and aggregate library projects are distinct and behave very differently: an aggregate project produces no artifact and can only be used as a root, while an aggregate library project produces a library and can be imported like any other library project. Additionally, aggregate library projects cannot set default external variable values via the `External` attribute.

An aggregate project groups several independent project subtrees so they can all be built with a single GPRbuild invocation. Each entry in `Project_Files` is the root of one such subtree; the aggregate produces no build artifact of its own. It can also set default values for external variables (see *Scenarios*) that apply uniformly to all constituent subtrees via the `External` attribute.

Note

An aggregate project can only be used as a root project or as a constituent of another aggregate project. It cannot be imported via a `with` clause by any standard or library project.

9.1 Basic syntax

```
aggregate project All is
   for Project_Files use ("subsystem_a/a.gpr",
                          "subsystem_b/b.gpr",
                          "subsystem_c/c.gpr");
   for External ("BUILD") use "release";
end All;
```

`Project_Files` is mandatory and lists the roots of constituent subtrees as paths to `.gpr` files. Paths are relative to the aggregate project file, or absolute; they are not searched via the project search path. `External` sets a default value for an external variable visible to all constituent subtrees; an explicit `-X` switch on the command line takes precedence and can still override it.

Glob patterns are supported in `Project_Files`:

```
aggregate project All is
   for Project_Files use ("**/*.gpr");  --  all subtree roots recursively
end All;
```

The ** pattern searches subdirectories recursively and may only appear at the last position in the directory part of the path.

Constituent entries may themselves be aggregate projects, allowing nested groupings.

9.2 Use cases

9.2.1 Building all outputs from multiple subtrees

A common situation is a set of independent subtrees - some producing executables, some producing libraries - that need to be built together. Without an aggregate project, each subtree root requires a separate GPRbuild invocation. Grouping them in an aggregate builds everything in a single parallel invocation:

```
aggregate project All is
   for Project_Files use ("apps/app_a/app_a.gpr",
                          "apps/app_b/app_b.gpr",
                          "libs/util/util.gpr");
end All;
```

GPRbuild deduplicates any sources shared across subtrees and schedules all compilations, binds, and links in parallel.

9.2.2 Setting the build environment

The `External` and `Project_Path` attributes let an aggregate project fix the build environment for all constituent subtrees, replacing a long sequence of `-X` and `-aP` command-line switches:

```
aggregate project All is
   for Project_Files use ("a.gpr", "b.gpr");
   for Project_Path  use ("../shared/projects");
   for External ("BUILD") use "release";

   package Builder is
      for Global_Compilation_Switches ("Ada") use ("-g");
   end Builder;
end All;
```

`Project_Path` adds directories searched when resolving `with` clauses inside constituent subtrees (it does not affect the resolution of `Project_Files` paths themselves).

9.3 Restrictions

- An aggregate project may only `with` **abstract** projects (to share attribute values). It cannot `with` standard or library projects.
- An aggregate project cannot be extended.
- Source-related attributes (`Source_Dirs`, `Languages`, `Main`, etc.) and compilation packages (`Compiler`, `Binder`, `Linker`, `Naming`) are not permitted. Only the `Builder` package is allowed.

- `Object_Dir` is permitted. Although an aggregate project produces no build artifacts, `Object_Dir` provides a location for the build system to store incremental build data outside the source tree or any VCS-controlled directory.

CHAPTER

TEN

WORKING WITH TOOLS

The GPR tool suite includes several companion tools that share the same project model as GPRbuild. Tools that operate on existing project trees all accept the standard project options (`-P`, `-X`, `-aP`). A separate group of tools generates GPR project files as output rather than consuming them.

Each tool is fully documented in the *GPR Reference Manual*; this chapter provides a task-oriented overview and the most commonly used options.

10.1 Tools that operate on project trees

10.1.1 gprclean

`gprclean` removes the build artifacts produced by GPRbuild: object files, ALI files, libraries, executables, and binder-generated files. It reads the same project tree to know exactly what was produced and where.

```
$ gprclean -P myproject.gpr        # clean root project only
$ gprclean -P myproject.gpr -r     # clean entire project tree
```

Key options:

`-r`
: Clean all projects in the tree, not just the root.

`-c`
: Delete only compiler-generated files (object files, ALI files). Skip executables and libraries.

`-n`
: Dry run: list the files that would be deleted without removing them.

`-p`
: Remove empty object, library, and executable directories after cleaning.

See the *GPR Reference Manual*, chapter *gprclean*, for the complete option reference.

10.1.2 gprinstall

`gprinstall` copies build results into a target installation prefix. It records every installed file in a manifest, enabling precise uninstall later.

```
$ gprinstall -P mylib.gpr --prefix=/usr/local
$ gprinstall -P mylib.gpr --prefix=/usr/local --uninstall
$ gprinstall --list
```

Key options:

--prefix=<dir>
Root installation directory. Defaults to the active toolchain prefix.

--mode=<dev|usage>
`dev` (default) installs sources, ALI files, and libraries for use as a development dependency. `usage` installs only what end users need (shared libraries, executables).

--build-name=<name>
Tag the installation with a build variant name (e.g. `debug`, `production`). Allows multiple variants to coexist under the same prefix.

-r
Install all imported projects, not just the root.

--uninstall
Remove files recorded in the manifest for the named project.

--list
Print all installed packages found under the prefix.

See the *GPR Reference Manual*, chapter *gprinstall*, for the complete option reference.

10.1.3 gprls

`gprls` lists the sources, units, objects, and dependencies of a project tree. It reads the build database to report the up-to-date status of each artifact.

```
$ gprls -P myproject.gpr -s        # list source files
$ gprls -P myproject.gpr -o        # list object files
$ gprls -P myproject.gpr -d        # list dependencies with status
```

Key options:

-s
Print source file for each compilation unit.

-o
Print object file for each compilation unit.

-u
Print unit name for each Ada compilation unit.

-d
List source file dependencies with their up-to-date status (`OK` or `DIF`).

-U
Include sources from all projects in the tree, not just the root.

--closure
Compute the transitive compilation closure of the named source files.

See the *GPR Reference Manual*, chapter *gprls*, for the complete option reference.

10.1.4 gprinspect

`gprinspect` is a diagnostic tool that loads a project tree and displays its structure: project relationships, source directories, attributes, packages, variables, and type definitions. It is useful for understanding how a project tree is resolved and for debugging unexpected attribute values.

```
$ gprinspect -P myproject.gpr --all -r
```

Key options:

--display=<textual|json|json-compact>
: Output format. `textual` (default) is human-readable; `json` is suited for tool integration.

-r
: Display all projects in the tree, not just the root.

--all
: Display attributes, packages, variables, and type definitions.

-c
: Include attributes inherited from the active configuration project.

See the *GPR Reference Manual*, chapter *gprinspect*, for the complete option reference.

10.2 Tools that generate project files

Unlike the tools above, the following tools produce GPR files as their output rather than loading and acting on an existing project tree.

10.2.1 gprconfig

`gprconfig` probes the host for available compilers and generates a configuration project (`.cgpr`) describing the selected toolchains to all GPR tools. It can be run interactively for guided toolchain selection, or in batch mode for scripted workflows, producing a persistent configuration project as an alternative to passing `--autoconf` to a GPR tool.

```
$ gprconfig                                # interactive selection
$ gprconfig --batch --config=Ada           # non-interactive, Ada compiler only
$ gprconfig --batch --config=Ada --config=C -o my.cgpr
```

Key options:

--target=<name>
: Select compilers for the given target. Defaults to native. Use `all` to list compilers for all known targets.

--batch
: Non-interactive mode; compiler selection is driven entirely by `--config` options.

--config=<selector>
: Pre-select a compiler. The selector is a comma-separated list of fields: `language,version,runtime,path,name`. Only the language field is required; trailing empty fields may be omitted. May be repeated (at most once per language).

  ```
  $ gprconfig --batch --config=Ada                    # any Ada compiler
  $ gprconfig --batch --config=Ada,,native            # Ada, native runtime
  $ gprconfig --batch --config=Ada,14,,/opt/gnat/bin  # Ada 14, specific path
  ```

-o <file>
: Write the configuration project to the given file. Defaults to `default.cgpr` in the GPRbuild configuration directory.

--show-targets
: Print the list of targets for which compilers are available, then exit.

See the *GPR Reference Manual*, chapter *gprconfig*, for the complete option reference.

10.2.2 gprname

`gprname` scans source directories for Ada source files, identifies the Ada units they contain, and generates a GPR project file with a `Naming` package that maps each unit to its source file. It is the standard tool for setting up a project whose sources do not follow the default GNAT naming conventions.

```
$ gprname -P myproject.gpr -d src/
```

This creates (or updates) `myproject.gpr` and a companion `myproject_naming.gpr` containing the `Naming` package. A source list file `myproject_source_list.txt` is also written.

Key options:

-d <dir>
: Add a source directory to scan. Append `/**` to search the directory and all its subdirectories recursively. May be repeated.

-f <pattern>
: Add a filename pattern for C sources.

-x <pattern>
: Exclude files matching a pattern from Ada source scanning.

--and
: Begin a new section with different source directories or patterns.

See the *GPR Reference Manual*, chapter *gprname*, for the complete option reference.

CHAPTER

ELEVEN

BEST PRACTICES

This chapter collects conventions and recommendations that apply across all project types.

11.1 Project layout

Keep one project file per component. A component is a unit of reuse: a library that other projects can import, or an executable with its own source tree. Mixing unrelated components in a single project makes incremental builds coarser and reuse harder.

Use dedicated directories for sources, objects, and outputs, and keep them separate from each other:

```
mylib/
├── mylib.gpr
├── src/
├── obj/
└── lib/
```

Never place generated artifacts (object files, libraries, executables) inside the source tree. Mixing them with sources complicates version control, as generated files appear as untracked changes and require explicit exclusion.

To keep the source repository completely free of generated files, `--relocate-build-tree` redirects all build artifacts to a directory outside the repository at invocation time, regardless of the relative paths declared in the project file (see *Building Executables*).

11.2 Scenario variables

Keep the number of scenario variables small. Each variable multiplies the number of distinct project configurations; a tree with four binary variables has sixteen possible configurations to reason about.

For variable names, two strategies work well depending on the context:

- **Consistent names** across all projects in a tree, so that a single `-X` switch on the command line applies everywhere. The conventional names are `BUILD` (debug/release), `TARGET` (cross-compilation target), and `RUNTIME` (Ada runtime variant). Use this approach for variables that genuinely apply to the whole tree.
- **Distinctive names** for variables that are local to a specific component and should not accidentally match a variable in another project. A component-specific prefix (e.g. `MYLIB_FLAVOR`) prevents unintended coupling.

Declare scenario types and variables in a shared `abstract` project and import it everywhere (see *Scenarios*). This guarantees that all projects agree on the accepted value set and avoids load errors when a value valid for one project is rejected by another.

11.3 Object and output directories

Never share `Object_Dir` or `Library_Dir` between two projects. If two projects write build artifacts to the same directory, a filename collision is reported as an error.

When scenario variables produce multiple build configurations from the same project file, include the variable value in the directory path so each configuration writes to its own location:

```
for Object_Dir  use "obj/" & Build;
for Library_Dir use "lib/" & Build;
```

If two configurations share the same `Object_Dir`, switching between them forces a full rebuild because all objects from the previous configuration are present and must be replaced.

11.4 Multi-project hygiene

Declare shared compiler switches, scenario variables, and naming conventions in a single `abstract` project and import it from every project in the tree. This is the primary mechanism for keeping a large project tree consistent.

Use `Compiler'Switches` (in individual projects) rather than `Builder'Global_Compilation_Switches` for project-local settings. `Global_Compilation_Switches` propagates to all imported projects, which is rarely the intended effect and makes it harder to reason about what flags are applied to each source.

Avoid `limited with` unless genuinely necessary (see *Organizing Multi-Project Systems*). A need for it often signals that two components should be refactored into a shared lower-level dependency rather than kept mutually visible.

11.5 Performance

For large project trees with many independent components, use an aggregate project as the root. Shared sources are deduplicated across subtrees, and GPRbuild maximizes parallelism across the whole group in a single invocation (see *Aggregate Projects*).

Declare `Object_Dir` on aggregate projects so that incremental build data is stored in a well-known location outside the source tree and outside any VCS-controlled directory.

For stable subcomponents that change infrequently, consider installing them with `gprinstall` (see *Working with Tools*) and referencing them from the project search path. Installed projects are treated as *externally built* - their sources are considered read-only and their pre-built artifacts are consumed directly - which significantly reduces build times for large trees.

Consider using a static configuration project via `--config` or `--autoconf`. The default automatic configuration probes the host for many potential toolchain candidates and this may generate a significant overhead. Note that with a static configuration, it must be updated whenever the toolchain changes or new languages are used in the project.

CHAPTER

TWELVE

GNU FREE DOCUMENTATION LICENSE

Version 1.3, 3 November 2008

INDEX

T

W