CS111 Introduction to Systematic Programming

Compiling Ada Packages

In Unit 17 of the ISP courses Ada packages are introduced. Library packages such as `CS_Int_IO` and `CS_File_IO` are already compiled as are standard Ada packages such as `Ada.Text_IO`. However if users write their own packages, these need to be compiled before they can be used in other programs. For the purposes of illustration suppose that you have written an Ada package `MyPack` in the files `mypack.ads` (for the package specification) and `mypack.adb` (for the package body). To compile the package:

```
gcc -c mypack.adb
```

Note that it would be an error to attempt to use `gnatbl` to attempt to bind and link the package into a complete executable program.

Notes

The package specification and package body are checked for compatibility when the body is compiled and if inconsistencies are found, the compilation is aborted. Thus the package specification (`.ads` file) must be available when the package body is compiled.

With the Gnat Ada system if a package has both a specification and body only the body is compiled. However during development of the package it is recommended that the package specification (which is normally written before the body) is compiled to check for any syntax and semantic errors:

```
gcc -c -gnatc mypack.ads
```

The switch `-gnatc` means do syntax and semantic checking only. If it is left out when compiling a package specification `gcc` emits a message warning that no object code has been produced.

Some packages (such as `Common_Pack` in Unit 17) contain only type declarations and contain no executable code. Such packages have no body, that is no `.adb` file. In such cases it is also prudent to check the syntax of the specification by doing:

```
gcc -c -gnatc common_pack.ads
```

Occasionally a package may have no body, but may still contain executable code; `CS_Int_IO` and `CS_Flt_IO` are two such examples. In such cases the package specification must be compiled **without** the switch `-gnatc` (as object code will normally be produced)

```
gcc -c cs_int_io.ads
```

Using the Package

When you need to use the pack in a client program in the file `myprog.adb` (say) you insert the lines

```
WITH  MyPack;   USE   MyPack;        -- USE optional[1]
```

at the head of the file `myprog.adb` and then compile and link the program in the normal way:

```
gcc -c myprog.adb
gnatbl myprog.ali
```

or

```
gnatmake myprog.adb
```

Provided that the client program `myprog.adb` **and the package files** `mypack.ads` and `mypack.adb` **are in the same directory**, this will cause the file `myprog.adb` to be compiled and if this is successful the program will be linked with the library package `MyPack` (and any other imported packages) to form a complete executable program.

---

[1] If the `USE` clause is omitted then fully qualified names have to be used for the procedures, functions etc. which are provided by the package.

Notes
The package specification (`.ads` file) must be available before any client programs which use the package can be compiled. The compiler uses the package specification to check that the client program is calling the subprograms exported by the package with the correct number and type of parameters.

The compiled version of the package (`.o` and `.ali` files) must be available before the client program can be bound and linked with `gnatbl`. Note that the source file of the package body (`.adb` file) need not be available to the client at all.

If the package body is changed without altering the package specification (perhaps to correct a bug or to make the implementation more efficient), then client programs need not be re-compiled, however they must be re-linked if they are to use the new version of the package body.

If the package specification and the body are changed, any client programs which use them must be re-compiled and, of course, re-linked.

Building a Separate Library Directory
The simple procedure described above works **only when the client program and the package are in the same directory**.

When working on large programming projects it is customary to set up a separate Ada directory for each project. Suppose two projects needed to use the library package `MyPack`. Of course, we could place copies of `mypack.ads`, `mypack.ali` and `mypack.o` (the compiled forms of `mypack.adb`) in each project directory, but obviously this wastes disk space. Also there is the danger that if one version of `MyPack` is edited, there will be two (perhaps incompatible) versions of the package in different directories and this could easily cause confusion.

A better solution is to place the package in a separate directory `MyLibrary` (say), compile it and then make it available to the project directories by setting the environment variables `ADA_INCLUDE_PATH` and `ADA_OBJECTS_PATH` to add the new library to the Gnat search paths.

The variable `ADA_INCLUDE_PATH` specifies those directories that the compiler `gcc` should search for package specifications (`.ads` files).

`ADA_OBJECTS_PATH` specifies those directories which the binder `gnatbind` should search for Ada Library Information files (`.ali` files) and that the linker `gnatlink` (and `ld`) should search for object files (`.o` files).

The whole process would be as follows:

```
cd                            # move to your home directory
mkdir MyLibrary               # create a UNIX directory
mv mypack.ad[bs] MyLibrary    # move the 2 source files to this directory
cd MyLibrary                  # make MyLibrary the current working directory
gcc -c -gnatc mypack.ads      # compile the spec. (to check syntax)
gcc -c mypack.adb             # compile the body

# add ~/MyLibrary to the existing Ada include and objects paths
setenv ADA_INCLUDE_PATH  ~/MyLibrary:$ADA_INCLUDE_PATH
setenv ADA_OBJECTS_PATH  ~/MyLibrary:$ADA_OBJECTS_PATH
                              # The set-up of the library is now complete
```

To make the library automatically available to the Gnat Ada system in future login sessions, add the following two lines:

```
setenv ADA_INCLUDE_PATH  ~/MyLibrary:$ADA_INCLUDE_PATH
setenv ADA_OBJECTS_PATH  ~/MyLibrary:$ADA_OBJECTS_PATH
```

at the end of your `.login` file.

To inspect the current values of the Ada include and object paths do

```
printenv ADA_INCLUDE_PATH
printenv ADA_OBJECTS_PATH
```

Location of the Ada Libraries
The source and object files for the special CS libraries (`CS_Int_IO`, etc.) used in the ISP course may be found in the directory:

```
/usr/local/staffstore/CSAdaLib
```

The source files for the standard Ada libraries (`Ada.Text_IO`, etc.) may be found in the directory:

```
/usr/local/gnat3.13p/lib/gcc-lib/sparc-sun-solaris2.7/2.8.1/adainclude
```

The Ada Library Information for standard Ada libraries (`Ada.Text_IO`, etc.) may be found in the directory:

```
/usr/local/gnat3.13p/lib/gcc-lib/sparc-sun-solaris2.7/2.8.1/adalib
```

The corresponding object files are in a UNIX object archive file `libgnat.a` in this directory.

For a list of mappings of package names in the standard Ada library to source file names, see the file

```
/usr/local/staffstore/CSAdaLib/gnat_filenames.txt
```

Using `gnatmake`

So far we have only used `gnatmake` as a convenient short-cut to compile, bind and link instead of typing separate `gcc` and `gnatbl` commands. However `gnatmake` is in fact much more powerful.

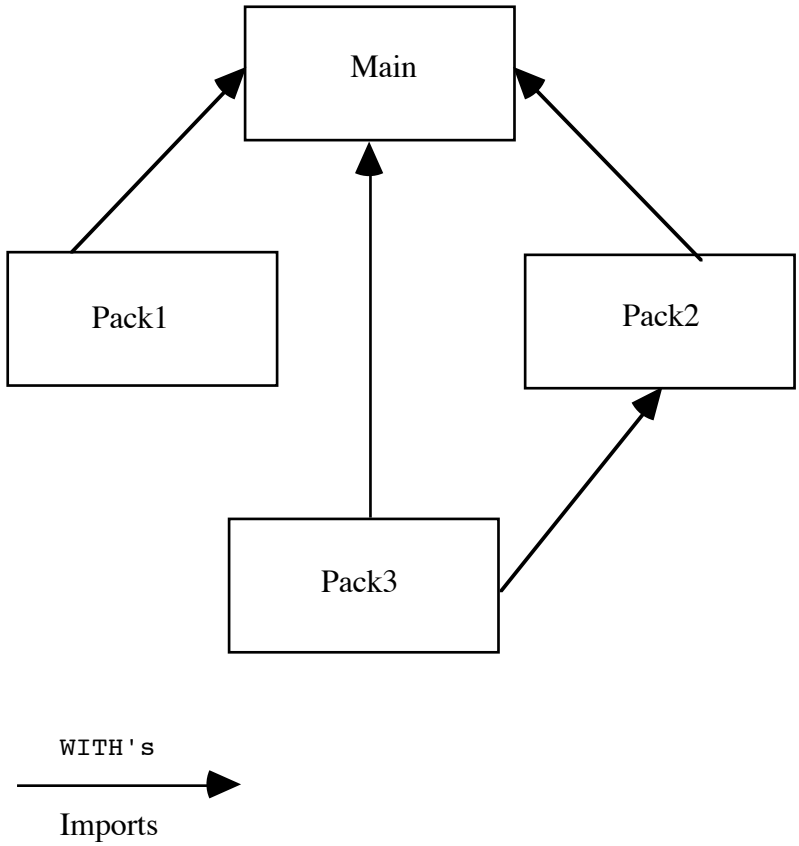
When a large program is developed it is normally decomposed into a considerable number of separate Ada packages (as described in Unit 17 of the ISP notes). During development those packages currently under development (and the main program) will need to be modified and recompiled regularly whilst packages which have not changed will not need recompilation.

If the recompilation process is initiated manually, there is a danger that some packages which ought to be recompiled will be forgotten (leading to an inconsistent and buggy system) and that packages which do not need recompiling are compiled unnecessarily (wasting time and computer resources).

The utility `gnatmake` avoids such problems by deducing which files need to be recompiled when the a multi-package system is rebuilt. `gnatmake` analyses which files need recompiling, that is those that have been modified since the last time the system was built and those that import packages (using `WITH`) whose specifications have changed since the last rebuild. Then it automatically does all the necessary recompilations and then links the latest compiled versions of all the required packages into the final executable program. `gnatmake` works by comparing the modification times of the source files (those with `.ads` or `.adb` extensions) with the modification times of the corresponding compiled files (with `.ali` and `.o` extensions). If the source file was modified more recently than its compiled version or if it depends on a package which has been modified, then it needs to be recompiled.

For example, suppose we have a main program in a file called `main.adb` and that this imports (i.e. `WITH`'s) three packages `Pack1`, `Pack2` and `Pack3` which are stored in files `pack1.ads`, `pack1.adb`, `pack2.ads`, `pack2.adb` and `pack3.ads` and `pack3.adb` respectively. Suppose also that the body of `Pack2` namely `pack2.adb` imports (i.e. `WITH`'s) `Pack3`, but that `Pack1` does not. If `Pack3`'s specification is changed then not only does `pack3.adb` need recompiling but so too do `main.adb` and `pack2.adb`. However `pack1.o` is 'up-to-date' and so `pack1.adb` does not need to be recompiled. After recompiling all necessary files, the latest `.o` files are

linked using information in the (latest) `.ali` files. However if only `Pack3`'s body is changed, then only `pack3.adb` needs recompiling and then the whole system is re-linked with the new `pack3.o` file.



To summarise:

> if a package specification is changed, the package body and any package or program that depends on it (any package/program that `WITH`'s the package) must be recompiled and relinked.
>
> if only the package body is changed, the package body must be recompiled and any program that depends on the package (any program that `WITH`'s the package) must be relinked, but need not be recompiled.

To rebuild the system using `gnatmake` all that is required is to issue the call

```
gnatmake main.adb
```

and all the necessary recompilations (and no others) will be performed and the system linked to produce an executable program called `main`.

File Naming Conventions

If `gnatmake` is to find all the necessary packages and work correctly the following file naming conventions **should** be used[2]:

> Package specifications and bodies must have the extensions `.ads` and `.adb` respectively.
>
> The basename of the files containing the specification and the body must be the same as the name of the package, but with all letters converted to lowercase[3].

---

[2] For ways of avoiding these restrictions consult the Gnat User Guide in the file `/usr/local/gnat3.13p/docs/gnat_ug.txt`

The file containing the main program should have the extension `.adb`

Thus the specification and body of a package `CS_File_IO` should be saved in files `cs_file_io.ads` and `cs_file_io.adb` respectively.

Notes
Both `gnatmake` and `gcc` issue **warning** messages if the main program is stored in a file with a basename that is not the same as the name of the main procedure (converted to lowercase). However these are only warning messages and the program will be compiled and linked correctly if the above stated conventions are followed.

The names of the source files for the standard Ada Libraries are abbreviated and do not follow the above conventions. For a list of mappings of package names in the standard Ada library to source file names, see the file

```
/usr/local/staffstore/CSAdaLib/gnat_filenames.txt
```

`gnatmake` Options
The behaviour of `gnatmake` may be controlled by supplying command-line options. Only a few of the available options will be described here; for more information consult the GNAT User Guide in the file `/usr/local/gnat3.13p/docs/gnat_ug.txt`[4]. The general form of a `gnatmake` command is :

```
gnatmake [options] filename [-cargs options] [-bargs options] [-largs options]
```

The parts above which are enclosed in square brackets are optional and may be omitted. This convention is used in all Unix manual pages.

General Options (placed before the filename)

`-c`	compile all necessary units, but do not bind and link.
`-f`	force recompilation of all units even if they are up to date.
`-g`	compile all necessary units and the bind file with debugging info. for `gdb` included.
`-n`	check whether any recompilations are needed, but don't actually compile, bind or link anything.
`-o name`	call the final program produced `name` rather than using the basename of the file being compiled.
`-v`	verbose mode, trace the actions of `gnatmake` as it proceeds -- quite instructive.
`-Adir`	make the compiler search the specified directory `dir` for `.ads` files -- an alternative to adding `dir` to `ADA_INCLUDE_PATH`.
`-Idir`	make the compiler, binder and linker search the specified directory `dir` for `.ads`, `.ali` and `.o` files -- an alternative to adding `dir` to both `ADA_INCLUDE_PATH` and `ADA_OBJECTS_PATH`.
`-Ldir`	make the binder and linker search the specified directory `dir` for `.ali` and `.o` files respectively -- an alternative to adding `dir` to `ADA_OBJECTS_PATH`.

Compiler Options (placed after `-cargs`)

`-g`	compile the Ada source file(s) with debugging info for `gdb` included
`-gnatc`	check syntax and semantics only, but do not produce object code
`-gnatl`	if the compilation fails, produce a long error listing with error messages embedded in the full source code

---

[3] Also any dots in the package name should be converted to underscores. Thus the spec. and body of a package called `Random.Float` should be saved in files `random_float.ads` and `random_float.adb` respectively.

[4] Warning -- this is a long document with around 100 pages in total. There is also a PostScript version of this document called `gnat_ug.ps` in the same directory.

`-gnatv` if the compilation fails, produce a verbose error listing with error messages and the offending lines of source code

Binder Options (placed after `-bargs`)

`-g` compile the bind file with debugging info for `gdb` included
`-Idir` make the binder search the specified directory `dir` for `.ali` files.

Linker Options (placed after `-largs`)
Sometimes object files (`.o` files) are stored in a UNIX archive file (`.a` file). The linker switch `-l` is then necessary:

`-lfile` make the linker search the object library archive file `libfile.a` for the required `.o` files.

If some modules of the system are written in a language other than Ada (usually either C or C++) or if the system uses non-standard C libraries (e.g. those for the X-windows tool-kit etc.), the `-L` linker switch will be needed:

`-Ldir` make the linker search the directory `dir` for object files and object library archives (`.o` and `.a` files).