Ada Programming

by Wikibooks contributors

Developed on **Wikibooks**,
the open-content textbooks collection

Principal authors:

- Martin Krischik (Contributions)
- Manuel Gómez (Contributions)
- Santiago Urueña (Contributions)
- C.K.W. Grein (Contributions)
- Bill Findlay (Contributions)
- Simon Wright (Contributions)
- John Oleszkiewicz (Contributions)
- Nicolas Kaiser (Contributions)
- Larry Luther (Contributions)
- Georg Bauhaus (Contributions)
- Samuel Tardieu (Contributions)
- Ludovic Brenta (Contributions)
- Ed Falis
- Pascal Obry
- Unnamed Hero (Contributions)
- B. Seidel (Contributions)
- Allen Lew (Contributions)

The current version of this Wikibook may be found at:
http://en.wikibooks.org/wiki/Ada_Programming

Table of Contents

1 Preface

Welcome to the **Ada Programming** tutorial at Wikibooks. This is the first Ada tutorial covering the Ada 2005 standard. If you are a beginner you will learn the latest standard - if you are a seasoned Ada user you can see what's new.

Current Development Stage for **Ada Programming** is "(Jul 27, 2005)". At this date, there are more than 200 pages in this book, which makes **Ada Programming** the largest of the programming wikibooks[1].

But still there is always room for improvement — do help us to expand **Ada Programming**. Even beginners will find areas to participate.

About Ada

Wikipedia has related information at ***Ada programming language***.

Ada is a programming language named after Augusta Ada King, Countess of Lovelace, which is suitable for all development needs.

Ada has built-in features that directly support structured, object-oriented, generic, distributed and concurrent programming.

Ada is a good choice for Rapid Application Development, Extreme Programming (XP), and Free Software development.

Programming in the large

Ada puts unique emphasis on, and provides strong support for, good software engineering practices that scale well to very large software systems (millions of lines of code, and very large development teams). The following language features are particularly relevant in this respect:

- An extremely strong, static and safe **type system**, which allows the programmer to construct powerful abstractions that reflect the real world, and allows the compiler to detect many logic errors before they become bugs.

- **Modularity**, whereby the compiler directly manages the construction of very large software systems from sources.

- **Information hiding**; the language separates interfaces from implementation, and provides fine-grained control over visibility.

- **Readability**, which helps programmers review and verify code. Ada favours the reader of the program over the writer, because a program is written once but read many times. For example, the syntax bans all ambiguous constructs, so there are no surprises, in accordance with the Tao of Programming's Law of Least Astonishment. (Some Ada programmers are reluctant to talk about

source code which is often cryptic; they prefer *program text* which is close to English prose.)

- **Portability**: the language definition allows compilers to differ only in a few controlled ways, and otherwise defines the semantics of programs very precisely; as a result, Ada source text is very portable across compilers and across target hardware platforms. Most often, the program can be recompiled without any changes[2].

- **Standardisation**: standards have been a goal and a prominent feature ever since the design of the language in the late 1970's. The first standard was published in 1980, just 3 years after design commenced. Ada compilers all support the exact same language; there are no dialects.

Consequences of these qualities are superior **reliability**, **reusability** and **maintainability**. For example, compared to programs written in C, programs written in Ada 83 contain ten times fewer bugs, and cost half as much to develop in the first place[3]. Ada shines even more in software maintenance, which often accounts for 80% of the total cost of development. With support for object-oriented programming, Ada95 brings even more cost benefits, although no serious study comparable to Zeigler's has been published.

Programming in the small

In addition to its support for good software engineering practices, which are applicable to general-purpose programming, Ada has powerful specialised features supporting low-level programming for real-time, safety-critical and embedded systems. Such features include, among others, machine code insertions, address arithmetic, low-level access to memory, control over bitwise representation of data, bit manipulations, and a well-defined, statically provable concurrent computing model called the Ravenscar Profile.

Other features include restrictions (it is possible to restrict which language features are accepted in a program) and features that help review and certify the object code generated by the compiler

Several vendors provide Ada compilers accompanied by minimal run-time kernels suitable for use in certified, life-critical applications. It is also possible to write Ada programs which require no run-time kernel at all.

It should come as no surprise that Ada is heavily used in the aerospace, defense, medical, railroad, and nuclear industries.

The Language Reference Manual

Wikipedia has related information at ***ISO 8652***.

The Ada Reference Manual (RM) is the official language definition. If you have a problem, if no one else can help, and you can find it, maybe you should read the RM (albeit often a bit cryptic for non-language-lawyers). For this reason, all complete (not draft) pages in **Ada Programming** contain links into the appropriate pages in the RM.

This tutorial covers *Ada Reference Manual — ISO/IEC 8652:1995(E) with Technical Corrigendum 1:2001 and Amendment 1:2007 — Language and Standard Libraries*, colloquially known as *Ada 2005* or just *Ada*.

You can browse the complete Reference Manual at http://www.adaic.com/standards/05rm/html/RM-TTL.html

There are two companion documents:

- The Annotated Reference Manual, an extended version of the RM aimed at compiler writers or other persons who want to know the fine details of the language.
- The Rationale for Ada 2005, an explanation of the features of the language.

The Ada Information Clearinghouse also offers the older Ada 83 and Ada 95 standards and companion documents.

The RM is a collective work under the control of Ada users. If you think you've found a problem in the RM, please report it to the Ada Conformity Assessment Authority. On this site, you can also see the list of "Ada Issues" raised by other people.

Ada Conformity Assessment Test Suite

Wikipedia has related information at ***ISO 18009***.

Unlike other programming languages, Ada compilers are officially tested, and only those which pass this test are accepted, for military and commercial work. This means that all Ada compilers behave (almost) the same, so you do not have to learn any dialects. But because the Ada standard allows the compiler writers to include some additions, you could learn a cool new feature only to find out that your favorite compiler does not support it....

Programming in Ada

Getting Started

Where to get a compiler, how to compile the source, all answered here:

- Basic Ada — ⊞(Sep 22, 2005), **Read Me First!**
- Finding and Installing Ada — ⊞(Sep 22, 2005)
- Building an Ada program — ⊞(Sep 22, 2005)

Language Features

These chapters look at the broader picture. They are more tutorial like and show how the keywords, operators and so forth work together.

- Control Structures — (Dec 14, 2005)
- Type system — (Jan 5, 2006)
- Strings — (Jan 5, 2006)
- Subprograms — (Jan 5, 2006)
- Packages — (Jan 5, 2006)
- Input Output — (Jan 5, 2006)
- Exceptions — (Jan 5, 2006)
- Generics — (Jan 5, 2006)
- Tasking — (Jan 5, 2006)
- Object Orientation — (Jan 5, 2006)
- Memory management — (Jan 5, 2006)
- New in Ada 2005 — (Jan 5, 2006)
- Containers — (Jan 5, 2006)
- Ada Programming Tips — (Jan 5, 2006)

Computer Programming

The following articles are Ada adaptations from articles of the Computer programming book. The texts of these articles are language neutral but the examples are all Ada.

- Algorithms
  - Chapter 1
- Chapter 6
- Knuth-Morris-Pratt pattern matcher
- Binary search
- Error handling
- Function overloading
- Mathematical calculations
- Statements
  - Control
- Expressions
- Variables

Language Summary

Within the following chapters we look at foundations of Ada. These chapters may be used for reference of a particular keyword, delimiter, operator and so forth.

- Lexical elements — (Dec 14, 2005)
  - Keywords — (Jan 10, 2005)
- Delimiters — (Jan 10, 2005)
- Operators — (Jan 10, 2005)
- Attributes — (Jan 10, 2005)
- Pragmas — (Jan 10, 2005)

Predefined Language Libraries

The library which comes with Ada in general and GNAT in particular. Ada's built-in library is

extensive and well structured. These chapters serve as a reference for Ada's built-in libraries.

- Standard — (Sep 22, 2005)
- Ada — (Sep 22, 2005)
- Interfaces — (Sep 22, 2005)
- System — (Sep 22, 2005)
- GNAT — (Sep 22, 2005)

Other Language Libraries

Other libraries which are not part of the standard but freely available.

- Libraries
  - Multi Purpose
- Container Libraries
- GUI Libraries
- Distributed Objects
- Database
- Web Programming
- Input/Output
- Platform
  - Programming Ada 95 in Linux
- Programming Ada 95 in Windows
- Programming Ada 95 in Virtual Machines (Java, .NET)

External resources

- Open-source portals
- Web Tutorials

Source Code

The Source from the Book is available for download and online browsing. The latter allows "drill down", meaning that you can follow the links right down to the package bodies in the Ada runtime library.

References

1. ↑ wikistats, Category:Ada Programming or /All Chapters)
2. ↑ paper by Eurocontrol (PDF, 160 kB) on Portability
3. ↑ Study by Stephen F. Zeigler on Ada vs. C

See also

Resources

- AdaWikiRing — A list of Ada Wikis
- AdaPower — Ada Tools and Resources
- Ada World — A brand new Ada related website
- Ada Answers — Building better software with Ada
- comp.lang.ada (Google groups) — Newsgroup.

Manuals and guides

- Annotated Ada Reference Manual
- Ada 95 Style Guide
- Guide for the use of the Ada programming language in high integrity systems (ISO/IEC TR 15942:2000) — Analytic-based coding standards. ISO Freely Available Standards [1]
- Ada 95 Trustworthiness Study: Guidance on the Use of Ada 95 in the Development of High Integrity Systems Version 2.0 (1997) & Appendices — Engineering-based guidelines and coding standards.

Associations

- ACM SIGAda — ACM Special Interest Group on Ada
- Ada-Europe
- Ada Germany

Free online books/courses

- Computer-Books.us — Online Ada books
- Online Ada Book
- Book: Ada Distilled
- Ada-95: A guide for C and C++ programmers
- Ada in Action
- Introducing Ada95
- Learn Ada on the Web
- Quick Ada
- Introducing Ada
- Ada for Software Engineers — Free textbook originally published by John Wiley.
- Ada 95: The Craft of Object-Oriented Programming — Free textbook originally published by Prentice Hall.

Authors and contributors

This Wikibook has been written by:

- Martin Krischik (Contributions)
- Manuel Gómez (Contributions)
- Santiago Urueña (Contributions)
- C.K.W. Grein (Contributions)
- Bill Findlay (Contributions)
- B. Seidel (Contributions)
- Simon Wright (Contributions)
- Allen Lew (Contributions)
- John Oleszkiewicz (Contributions)
- Nicolas Kaiser (Contributions)
- Larry Luther (Contributions)
- Georg Bauhaus (Contributions)
- Samuel Tardieu (Contributions)
- Ludovic Brenta (Contributions)
- Ed Falis
- Pascal Obry
- Unnamed Hero (Contributions)

If you wish to contribute as well you should read Contributing and join us at the Contributors lounge.

2 Basic

"Hello, world!" programs

"Hello, world!"

A common example of a language's syntax is the Hello world program. Here a straight-forward Ada Implementation:

File: hello_world_1.adb (view, plain text, download page, browse all)

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

The **with** statement adds the package `Ada.Text_IO` to the program. This package comes with every Ada compiler and contains all functionality needed for textual Input/Output. The **with** statement makes the declarations of `Ada.Text_IO` available to procedure `Hello`. This includes the types declared in `Ada.Text_IO`, the subprograms of `Ada.Text_IO` and everything else that is declared in `Ada.Text_IO` for public use. In Ada, packages can be used as toolboxes. `Text_IO` provides a collection of tools for textual input and output in one easy-to-access module. Here is a partial glimpse at package Ada.Text_IO:

```
package Ada.Text_IO is

   type File_Type is limited private;

   --  more stuff

   procedure Open(File : in out File_Type;
                  Mode : File_Mode;
                  Name : String;
                  Form : String := "");

   --  more stuff

   procedure Put_Line (Item : String);

   --  more stuff

end Ada.Text_IO;
```

Next in the program we declare a main procedure. An Ada main procedure does not need to be called "main". Any simple name is fine so here it is *Hello*. Compilers might allow procedures or functions to be used as main subprograms. [4]

The call on `Ada.Text_IO.Put_Line` writes the text "Hello World" to the current output file.

A **with** clause makes the content of a package *visible by selection*: we need to prefix the procedure name `Put_Line` from the `Text_IO` package with its full package name `Ada.Text_IO`. If you need procedures from a package more often some form of shortcut is needed. There are two options open:

"Hello, world!" with renames

By renaming a package it is possible to give a shorter alias to any package name.[5] This reduces the typing involved while still keeping some of the readability.

File: hello_world_2.adb (view, plain text, download page, browse all)

```
with Ada.Text_IO;
procedure Hello is
   package IO renames Ada.Text_IO;
begin
   IO.Put_Line("Hello, world!");
   IO.New_Line;
   IO.Put_Line("I am an Ada program with package rename.");
end Hello;
```

"Hello, world!" with use

The **use** clause makes all the content of a package directly visible. It allows even less typing but removes some of the readability. One suggested "rule of thumb": **use** for the most used package and **renames** for all other packages. You might have another rule (for example, always **use** Ada.Text_IO, never **use** anything else).

File: hello_world_3.adb (view, plain text, download page, browse all)

```
with Ada.Text_IO;
procedure Hello is
   use Ada.Text_IO;
begin
   Put_Line("Hello, world!");
   New_Line;
   Put_Line("I am an Ada program with package use.");
end Hello;
```

use can be used for packages and in the form of **use type** for types. **use type** makes only the operators of the given type directly visible but not any other operations on the type.

FAQ: Why is "Hello, world!" so big?

Ada beginners frequently ask how it can be that such a simple program as "Hello, world!" results in such a large executable. The reason has nothing to do with Ada but can usually be found in the compiler and linker options used — or better — not used.

Standard behavior for Ada compilers — or good compilers in general — is not to create the best code possible but to be optimized for ease of use. This is done to not frighten away potential new users by providing a system which does not work "out of the box".

The GNAT project files, which you can download alongside the example programs, use better tuned compiler, binder and linker options. If you use those your "Hello, world!" will be a lot smaller:

```
 32K ./Linux-i686-Debug/hello_world_1
8,0K ./Linux-i686-Release/hello_world_1
 36K ./Linux-x86_64-Debug/hello_world_1
 12K ./Linux-x86_64-Release/hello_world_1
1,1M ./Windows_NT-i686-Debug/hello_world_1.exe
 16K ./Windows_NT-i686-Release/hello_world_1.exe
 32K ./VMS-AXP-Debug/hello_world_1.exe
```

```
 12K ./VMS-AXP-Release/hello_world_1.exe
```

For comparison the sizes fora plain **gnat make** compile:

```
497K hello_world_1 (Linux i686)
500K hello_world_1 (Linux x86_64)
1,5M hello_world_1.exe (Windows_NT i686)
589K hello_world_1.exe (VMS AXP)
```

Worth mentioning is that hello_world (Ada,C,C++) compiled with GNAT/MSVC 7.1/GCC(C) all produces executables with approximately the same size given comparable optimisation and linker methods.

Things to look out for

It will help to be prepared to spot a number of significant features of Ada that are important for learning its syntax and semantics.

Comb Format

There is a *comb format* in all the control structures and module structures. See the following examples for the *comb format*. You don't have to understand what the examples do yet - just look for the similarities in layout.

```
if Boolean expression then
   statements
elsif Boolean expression then
   statements
else
   statements
end if;

declare
   declarations
begin
   statements
exception
   handlers
end;

procedure P (parameters : in out type) is
   declarations
begin
   statements
exception
   handlers
end P;

function F (parameters : in type) return type is
   declarations
begin
   statements
exception
   handlers
end F;

type R (discriminant : type) is
   record
      declarations
   end record;

type C (discriminant : type) is new parent class declaration
                              and interface declarations -- (Ada 2005)
```

```
with
   record
      declarations
   end record;


package P is
   declarations
private
   declarations
end P;


generic
   declarations
package P is
   declarations
private
   declarations
end P;


generic
   declarations
procedure P (parameters : in out type);
```

Note that semicolons consistently terminate statements and declarations; the empty line (or a semicolon alone) is not a valid statement: the null statement is

```
null;
```

Notes

1. ↑ wikistats, Category:Ada Programming or /All Chapters)
2. ↑ paper by Eurocontrol (PDF, 160 kB) on Portability
3. ↑ Study by Stephen F. Zeigler on Ada vs. C
4. ↑ Main subprograms may even have parameters; it is implementation-defined what kinds of subprograms can be used as main subprograms. The reference manual explains the details in 10.2 LRM 10.2(29) (Annotated): "…, an implementation is required to support all main subprograms that are public parameterless library procedures." *Library* means not nested in another subprogram, for example, and other things that needn't concern us now.
5. ↑ **renames** can also be used for procedures, functions, variables, array elements. It can not be used for types - a type rename can be accomplished with **subtype**.

Type and subtype

There is an important distinction between **type** and **subtype**: a type is given by a set of values and their operations. A subtype is given by a type, and a *constraint* that limits the set of values. Values are always of a type. Objects (constants and variables) are of a subtype. This generalizes, clarifies and systematizes a relationship, e.g. between *Integer* and 1..100, that is handled *ad hoc* in the semantics of Pascal.

Constrained types and unconstrained types

There is an important distinction between *constrained* types and *unconstrained* types. An unconstrained type has one or more free parameters that affect its size or shape. A constrained type fixes the values of these parameters and so determines its size and shape. Loosely speaking, objects must be of a constrained type, but formal parameters may be of an unconstrained type (they adopt the constraint of

any corresponding actual parameter). This solves the problem of array parameters in Pascal (among other things).

Dynamic types

Where values in Pascal or C must be static (e.g. the subscript bounds of an array) they may be dynamic in Ada. However, static expressions are required in certain cases where dynamic evaluation would not permit a reasonable implementation (e.g. in setting the number of digits of precision of a floating point type).

Separation of concerns

Ada consistently supports a separation of interface and mechanism. You can see this in the format of a package, which separates its declaration from its body; and in the concept of a private type, whose representation in terms of Ada data structures is inaccessible outside the scope containing its definition.

3 Installing

Ada compilers are available from several vendors, on a variety of host and target platforms. The Ada Resource Association maintains a list of available compilers.

Below is an alphabetical list of available compilers with additional comments.

This list is incomplete. Please extend it as necessary.

AdaMagic from SofCheck

SofCheck produces an Ada 95 front-end that can be plugged into a code generating back-end to produce a full compiler. This front-end is offered for licensing to compiler vendors.

Based on this front-end, SofCheck offers:

- AdaMagic, an Ada-to-C translator
- AppletMagic, an Ada-to-Java bytecode compiler

Costs money; non-free. See http://www.sofcheck.com.

AdaMULTI from Green Hills Software

Green Hills Software sells development environments for multiple languages and multiple targets, primarily to embedded software developers.

Languages supported	Ada 83, Ada 95, C, C++, FORTRAN
License for the run-time library	proprietary
Native platforms	GNU/Linux on i386, Microsoft Windows on i386, and Solaris on SPARC
Cross platforms	INTEGRITY, INTEGRITY-178B and velOSity from Green Hills; VxWorks from Wind River; several bare board targets. Safety-critical GMART and GSTART run-time libraries certified to DO-178B level A.
Available from	http://www.ghs.com
Add-ons included	IDE, debugger, TimeMachine, integration with various version control systems, source browsers, other utilities

GHS claims to make great efforts to ensure that their compilers produce the most efficient code and often cites the EEMBC benchmark results as evidence, since many of the results published by chip manufacturers use GHS compilers to show their silicon in the best light, although these benchmarks are not Ada specific.

GHS has no publicly announced plans to support the new Ada standard published in 2007 but they do continue to actively market and develop their existing Ada products.

DEC Ada from HP

DEC Ada is an Ada 83 compiler for VMS. While "DEC Ada" is probably the name most users know, the compiler is now called "HP Ada". It had previously been known also by names of "VAX Ada" and "Compaq Ada".

- Ada for OpenVMS Alpha Installation Guide (PDF)
- Ada for OpenVMS VAX Installation Guide (PDF)

GNAT, the GNU Ada Compiler from AdaCore and the Free Software Foundation

GNAT is the free GNU Ada compiler, which is part of the GNU Compiler Collection. It is the only Ada compiler that supports all of the optional annexes of the language standard. The original authors formed the company AdaCore to offer professional support, consulting, training and custom development services. It is thus possible to obtain GNAT from many different sources, detailed below.

GNAT is always licensed under the terms of the GNU General Public License

However, the run-time library uses either the GPL, or the GNAT Modified GPL, depending on where you obtain it from.

Several optional add-ons are available from various places:

- ASIS, the Ada Semantic Interface Specification, is a library that allows Ada programs to examine and manipulate other Ada programs.
- FLORIST is a library that provides a POSIX programming interface to the operating system.
- GDB, the GNU Debugger, with Ada extensions.
- GLADE implements Annex E, the Distributed Systems Annex. With it, one can write distributed programs in Ada, where partitions of the program running on different computers communicate over the network with one another and with shared objects.
- GPS, the GNAT Programming Studio, is a full-featured integrated development environment, written in Ada. It allows you to code in Ada, C and C++.

Many Free Software libraries are also available.

GNAT GPL Edition

This is a source and binary release from AdaCore, intended for use by Free Software developers only. If you want to distribute your binary programs linked with the GPL run-time library, then you must do so under terms compatible with the GNU General Public License.

Languages supported	Ada 83, Ada 95, Ada 2005, C
License for the run-time library	pure GPL

Native platforms	GNU/Linux on i386, Microsoft Windows on i386, and Darwin (Mac OS X) on PowerPC
Cross platforms	none.
Compiler back-end	GCC 4.1.1
Available from	http://libre.adacore.com (requires free registration).
Add-ons included	GDB, GPS, GtkAda in source and binary form; many more in source-only form.

GNAT Modified GPL releases

With these releases of GNAT, you can distribute your programs in binary form under licensing terms of your own choosing; you are not bound by the GPL.

GNAT 3.15p

This is the last public release of GNAT from AdaCore that uses the GNAT Modified General Public License.

GNAT 3.15p has passed the Ada Conformity Assessment Test Suite (ACATS). It was released in October 2002.

The binary distribution from AdaCore also contains an Ada-aware version of the GNU Debugger (GDB), and a graphical front-end to GDB called the GNU Visual Debugger (GVD).

Languages supported	Ada 83, Ada 95, C
License for the run-time library	GNAT-modified GPL
Native platforms	GNU/Linux on i386 (with glibc 2.1 or later), Microsoft Windows on i386, OS/2 2.0 or later on i386, Solaris 2.5 or later on SPARC
Cross platforms	none.
Compiler back-end	GCC 2.8.1
Available from	ftp://ftp.cs.kuleuven.ac.be/pub/Ada-Belgium/mirrors/gnu-ada/3.15p/
Add-ons included	ASIS, Florist, GLADE, GDB, Gnatwin (on Windows only), GtkAda 1.2, GVD

GNAT Pro

GNAT Pro is the professional version of GNAT, offered as a subscription package by AdaCore. The package also includes professional consulting, training and maintenance services. AdaCore can provide custom versions of the compiler for native or cross development. For more information, see http://www.adacore.com.

Languages supported	Ada 83, Ada 95, Ada 2005, C, and optionally C++
License for the run-time library	GNAT-modified GPL
Native platforms	many, see http://www.adacore.com/home/gnatpro/configurations
Cross platforms	many, see http://www.adacore.com/home/gnatpro/configurations; even more on request

Compiler back-end	GCC 4.1.1
Available from	http://www.adacore.com by subscription (costs money)
Add-ons included	ASIS, Florist, GDB, GLADE, GPS, GtkAda, XML/Ada, and many more in source and, on request, binary form.

GCC

GNAT has been part of GCC since October 2001, and GCC uses the GNAT Modified General Public License for the Ada run-time library. The Free Software Foundation does not distribute binaries, only sources.

Most GNU/Linux distributions and several distributions for other platforms include prebuilt binaries; see below.

For technical reasons, we recommend against using the Ada compilers included in GCC 3.1, 3.2, 3.3 and 4.0. Instead, we recommend using GCC 3.4, 4.1 or 4.2, or one of the releases from AdaCore (3.15p, GPL Edition or Pro).

Since October 2003, AdaCore merge most of their changes from GNAT Pro into GCC during Stage 1; this happens once for each major release. In GCC 3.4, 4.0, 4.1 and 4.2, AdaCore have gradually added support for Ada 2005 as the language standard was finalised.

As of GCC 4.2, released on 2007-05-13:

Languages supported	Ada 83, Ada 95, Ada 2005, C, C++, Fortran 95, Java, Objective-C, Objective-C++
License for the run-time library	GNAT-modified GPL
Native platforms	none (source only)
Cross platforms	none (source only)
Compiler back-end	GCC 4.2
Available from	http://gcc.gnu.org in source only form
Add-ons included	none.

The GNU Ada Project

The GNU Ada Project provides source and binary packages of various GNAT versions for several operating systems, and, importantly, the scripts used to create the packages. This may be helpful if you plan to port the compiler to another platform or create a cross-compiler. For GNU/Linux users, there are instructions for building your own GNAT compiler at: http://ada.krischik.com/index.php/Articles/CompileGNAT

Both GPL and GMGPL versions of GNAT are available.

Languages supported	Ada 83, Ada 95, Ada 2005, C. (Some distributions also support Fortran 90, Java, Objective C and Objective C++)
License for the run-time library	pure or GNAT-modified GPL

Native platforms	Fedora Core 4 and 5, MS-DOS, OS/2, Solaris 10, SuSE 10 (more?)
Cross platforms	none.
Compiler back-end	GCC 2.8.1, 3.4, 4.0, 4.1 (various binary packages)
Available from	http://gnuada.sourceforge.net
Add-ons included	AdaBrowse, ASIS, Booch Components, Charles, GPS, GtkAda (more?)

GNAT for AVR microcontrollers

Rolf Ebert and others provide a version of GNAT configured as a cross-compiler to various AVR microcontrollers, as well as an experimental Ada run-time library suitable for use on the microcontrollers.

Languages supported	Ada 83, Ada 95, Ada 2005, C (more?)
License for the run-time library	GNAT-Modified GPL
Native platforms	GNU/Linux on i386
Cross platforms	Various AVR microcontrollers
Compiler back-end	GCC 3.4 (4.1 in progress)
Available from	http://avr-ada.sourceforge.net
Add-ons included	none

Prebuilt packages as part of larger distributions

Many distributions contain prebuilt binaries of GCC or various public releases of GNAT from AdaCore. Quality varies widely between distributions. The list of distributions below is in alphabetical oder. *(Please keep it that way.)*

AIDE (for Microsoft Windows)

AIDE - Ada Instant Development Environment is a complete one-click, just-works Ada distribution for Windows, consisting of GNAT, comprehensive documentation, tools and libraries. All are precompiled, and source code is also available. The installation procedure is particularly easy. AIDE is intended for beginners and teachers, but can also be used by advanced users.

Languages supported	Ada 83, Ada 95, C
License for the run-time library	GNAT-modified GPL
Native platforms	Microsoft Windows on i386
Cross platforms	none.
Compiler back-end	GCC 2.8.1
Available from	http://rochebruneadsl.free.fr/stephane/aide/
Add-ons included	ASIS, GDB, GPS, GtkAda (more?)

Blastwave (for Solaris on SPARC and x86)

Blastwave has binary packages of GCC 3.4.5 and 4.0.2 with Ada support. The package names are gcc3ada and gcc4ada respectively.

Languages supported	Ada 83, Ada 95, parts of Ada 2005, C, C++, Fortran 95, Java, Objective-C, Objective-C++
License for the run-time library	GNAT-modified GPL
Native platforms	Solaris and OpenSolaris on SPARC
Cross platforms	none.
Compiler back-end	GCC 3.4.5 and 4.0.2 (both available)
Available from	http://www.blastwave.org
Add-ons included	none (?)

Cygwin (for Microsoft Windows)

Cygwin, the Linux-like environment for Windows, also contains a version the GNAT Compiler. The Cygwin version of GNAT is older than the MinGW version and does not support DLLs and Multi-Threading (as of 11.2004).

Debian (GNU/Linux and GNU/kFreeBSD)

There is a Debian Policy for Ada (mirror), which tries to make Debian the best Ada development *and deployment* platform. The development platform includes the compiler and many libraries, prepackaged and integrated so as to be easy to use in any program. The deployment platform is the renowned *stable* distribution, which is suitable for mission-critical workloads and enjoys long life cycles, typically 3 to 4 years. Because Debian is a binary distribution, it is possible to deploy non-free, binary-only programs on it while enjoying all the benefits of a stable platform. Compiler choices are conservative for this reason, and the Policy mandates that all Ada programs and libraries be compiled with the same version of GNAT. This makes it possible to use all libraries in the same program. Debian separates run-time libraries from development packages, so that end users do not have to install the development system just to run a program.

The GNU Ada compiler can be installed on a Debian system with this command:

```
apt-get install gnat
```

This will also give you a list of related packages, which are likely to be useful for an Ada programmer.

Debian is unique in that it also allows programmers to use some of GNAT's internal components by means of two libraries: libgnatvsn (licensed under GNAT-Modified GPL) and libgnatprj (the project manager, licensed under pure GPL). Debian packages make use of these libraries.

Debian 3.1 *Sarge* released in June 2005 uses GNAT 3.15p as the default Ada compiler.

Debian 4.0 *Etch* released in April 2007 uses GCC 4.1 as the default Ada compiler.

	Sarge	**Etch**
Languages supported	Ada 83, Ada 95, C	Ada 83, Ada 95, parts of Ada 2005, C, C++, Fortran 95, Java, Objective-C, Objective-C++

License for the run-time library	GNAT-modified GPL	GNAT-modified GPL
Native platforms	Debian GNU/Linux on i386, sparc and powerpc	Debian GNU/Linux on alpha, amd64, hppa, i386, ia64, mips, mipsel, powerpc, s390, sparc; Debian GNU/kFreeBSD on i386
Cross platforms	none	none
Compiler back-end	GCC 2.8.1	GCC 4.1
Available from	http://www.debian.org	http://www.debian.org
Add-ons included	AdaBindX, AdaBrowse, AdaCGI, AdaSockets, ASIS, AUnit, AWS, Charles, Florist, GDB, GLADE, GNADE, GNAT Checker, GPS, GtkAda, OpenToken, TextTools, XML/Ada	AdaBrowse, AdaCGI, AdaControl, AdaSockets, ASIS, AUnit, AWS, Florist, GDB, GLADE, GNADE, GPS, GtkAda, OpenToken, TextTools, XML/Ada

The ADT plugin for Eclipse (see #ObjectAda_from_Aonix) can be used with GNAT as packaged for Debian Etch. Specify "/usr" as the toolchain path.

DJGPP (for MS-DOS)

DJGPP has GNAT as part of their GCC distribution.

DJGPP is a port of a comprehensive collection of GNU utilities to MS-DOS with 32-bit extensions, and is actively supported (as of 1.2005). It includes the whole GCC compiler collection, that now includes Ada. See the DJGPP website for installation instructions.

DJGPP programs run also in a DOS command box in Windows, as well as in native MS-DOS systems.

FreeBSD

FreeBSD's ports collection contains GNAT GPL 2005 Edition (package gnat-2005).

Languages supported	Ada 83, Ada 95, parts of Ada 2005, C
License for the run-time library	pure GPL
Native platforms	FreeBSD on i386 (more?)
Cross platforms	none.
Compiler back-end	GCC 3.4.6
Available from	http://www.freebsd.org
Add-ons included	none?

Gentoo GNU/Linux

The GNU Ada compiler can be installed on a Gentoo system using emerge:

```
emerge dev-lang/gnat
```

In contrast to Debian, Gentoo is primarily a source distribution, so many packages are available only in source form, and require the user to recompile them (using emerge).

Also in contrast to Debian, Gentoo supports several versions of GNAT in parallel on the same system. Be careful, because not all add-ons and libraries are available with all versions of GNAT.

Languages supported	Ada 83, Ada 95, Ada 2005, C (more?)
License for the run-time library	pure or GNAT-modified GPL (both available)
Native platforms	Gentoo GNU/Linux on amd64, powerpc and i386
Cross platforms	none.
Compiler back-end	GCC 3.4, 4.1 (various binary packages)
Available from	http://www.gentoo.org
Add-ons included	AdaBindX, AdaBroker, AdaDoc, AdaOpenGL, AdaSockets, ASIS, AUnit, Booch Components, CBind, Charles, Florist, GLADE, GPS, GtkAda, XML/Ada

GNAT for Macintosh (for Mac OS 9 and X)

GNAT for Macintosh provides a version of GNAT with Xcode integration and bindings.

Mandriva Linux

The GNU Ada compiler can be installed on a Mandriva system with this command:

```
urpmi gnat
```

MinGW (for Microsoft Windows)

MinGW - Minimalist GNU for Windows contains a version of the GNAT Compiler.

The following list should help you install. {I may have forgotten something - but this is wiki, just add to the list}

1. Install *MinGW-3.1.0-1.exe*.
    1. extract *binutils-2.15.91-20040904-1.tar.gz*.
2. extract *mingw-runtime-3.5.tar.gz*.
3. extract *gcc-core-3.4.2-20040916-1.tar.gz*.
4. extract *gcc-ada-3.4.2-20040916-1.tar.gz*.
5. extract *gcc-g++-3.4.2-20040916-1.tar.gz (Optional)*.
6. extract *gcc-g77-3.4.2-20040916-1.tar.gz (Optional)*.
7. extract *gcc-java-3.4.2-20040916-1.tar.gz (Optional)*.

8. extract *gcc-objc-3.4.2-20040916-1.tar.gz (Optional)*.
9. extract *w32api-3.1.tar.gz*.
10. Install *mingw32-make-3.80.0-3.exe (Optional)*.
11. Install *gdb-5.2.1-1.exe (Optional)*.
12. Install *MSYS-1.0.10.exe (Optional)*.
13. Install *msysDTK-1.0.1.exe (Optional)*.
    1. extract *msys-automake-1.8.2.tar.bz2 (Optional)*.
14. extract *msys-autoconf-2.59.tar.bz2 (Optional)*.
15. extract *msys-libtool-1.5.tar.bz2 (Optional)*.

I have made good experience in using *D:\MinGW* as target directory for all installations and extractions.

Also noteworthy is that the the Windows version for GNAT from Libre is also based on MinGW.

In gcc-3.4.2-release_notes.txt from MinGW site reads: *please* check that the files in the /lib/gcc/mingw32/3.4.2/adainclude and adalib directories are flagged as read-only. This attribute is necessary to prevent them from being deleted when using gnatclean to clean a project.

So be sure to do this.

SuSE Linux

All versions of SuSE Linux have a GNAT compiler included. SuSE versions 9.2 and higher also contains ASIS, Florist and GLADE libraries. The following two packages are needed:

```
gnat
gnat-runtime
```

For 64 bit system you will need the 32 bit compatibility packages as well:

```
gnat-32bit
gnat-runtime-32bit
```

ICC from Irvine Compiler Corporation

Irvine Compiler Corporation provide native and cross compilers for various platforms. The compiler and run-time system support development of certified, safety-critical software.

Costs money, but no-cost evaluation is possible on request. Royalty-free redistribution of the run-time system is allowed. Non-free. See http://www.irvine.com

Janus/Ada 83 and 95 from RR Software

RR Software offers native compilers for MS-DOS, Microsoft Windows and various Unix and Unix-like systems, and a library for Windows GUI programming called CLAW. There are academic, personal and professional editions, as well as support options. Cost money but relatively cheap; non-free.

ObjectAda from Aonix

Aonix offers native and cross compilers for various platforms, that cost money and are non-free. They come with an IDE, a debugger and a plug-in for Eclipse.

On Microsoft Windows and GNU/Linux on i386, Aonix offers two pricing models, at the customer's option: either a perpetual license fee with optional support, or just the yearly support fee: For Linux, that's $3000 for a single user or $12,000 for a 5-user service pack. See the full press release.

In addition, they offer "ObjectAda Special Edition": a no-cost evaluation version of ObjectAda that limits the size of programs that can be compiled with it, but is otherwise fully functional, with IDE and debugger. Free registration required; see http://www.aonix.com/oa_evaluation_request.html.

A recent contribution by Aonix is ADT for Eclipse. The *Ada Development Tools* add Ada language support to the Eclipse open source development platform. ADT can be used with Aonix compilers, and with GNAT. An open source vendor supported project is outlined for ADT at Eclipse. Codenamed *Hibachi* and showcased at the Ada Conference UK 2007 and during Ada-Europe 2007, the project has now been officially created.

Power Ada from OC Systems

See http://www.ocsystems.com/prod_powerada.html

Rational Apex from IBM Rational

Native and cross compilers, cost money, non-free. See http://www-306.ibm.com/software/awdtools/developer/ada/index.html

SCORE from DDC-I

Cross-compilers for embedded development, cost money, non-free. SCORE stands for Safety-Critical, Object-oriented, Real-time Embedded. See http://www.ddci.com.

XD Ada from SWEP-EDS

Cross-compilers for embedded development, cost money, non-free. Hosts include Alpha and VAX machines running OpenVMS. Targets include Motorola 68000 and MIL-STD-1750A processors. Ada 83 only. See http://www.swep-eds.com/XD%20Ada/Xd%20ada.htm

4 BUILDING

Ada programs are usually easier to build than programs written in other languages like C or C++, which frequently require a makefile. This is because an Ada source file already specifies the dependencies of its source unit. See the **with** keyword for further details.

Building an Ada program is not defined by the Reference Manual, so this process is absolutely dependent on the compiler. Usually the compiler kit includes a make tool which compiles a main program and all its dependencies, and links an executable file.

Building with various compilers

GNAT

With GNAT, you can run this command:

```
gnat make <your_unit_file>
```

If the file contains a procedure, gnatmake will generate an executable file with the procedure as main program. Otherwise, e.g. a package, gnatmake will compile the unit and all its dependencies.

GNAT command line

gnatmake can be written as one word `gnatmake` or two words `gnat make`. For a full list of gnat commands just type `gnat` without any command options. The output will look something like this:

```
GNAT 3.4.3 Copyright 1996-2004 Free Software Foundation, Inc.
List of available commands
GNAT BIND               gnatbind
GNAT CHOP               gnatchop
GNAT CLEAN              gnatclean
GNAT COMPILE            gnatmake -f -u -c
GNAT ELIM               gnatelim
GNAT FIND               gnatfind
GNAT KRUNCH             gnatkr
GNAT LINK               gnatlink
GNAT LIST               gnatls
GNAT MAKE               gnatmake
GNAT NAME               gnatname
GNAT PREPROCESS         gnatprep
GNAT PRETTY             gnatpp
GNAT STUB               gnatstub
GNAT XREF               gnatxref
Commands FIND, LIST, PRETTY, STUB and XREF accept project file switches -vPx, -Pprj and -Xnam=val
```

For further help on the option just type the command (one word or two words - as you like) without any command options.

GNAT IDE

The GNAT toolchain comes with an IDE (Integrated Development Environment) called GPS (GNAT Programming System). You need to download and install it separately. The GPS features a graphical user interface.

There are also GNAT plugins for Emacs (Ada Mode), KDevelop and Vim (Ada Mode) available.

Both Emacs and Vim Ada-Mode are maintained by The GNU Ada project.

GNAT with Xcode

Apple's free (gratis) IDE, Xcode, is included with every Macintosh but requires an explicit installation step from DVD-ROM or CD-ROM. It is also downloadable from http://developer.apple.com. Xcode uses the GNU Compiler Collection and thus supports Ada, GDB, etc., and also includes myriad tools for optimizing code which are unique to the Macintosh platform. However, GNAT must be installed separately as it is (as of 2005) not distributed as part of Xcode. Get the binary and/or sources at http://www.macada.org/, along with numerous tools and bindings including bindings to Apple's Carbon frameworks which allow the development of complete, "real" Mac programs, all in Ada.

Rational APEX

Rational APEX is a complete development environment comprising a language sensitive editor, compiler, debugger, coverage analyser, configuration management and much more. You normally work with APEX running a GUI.

APEX has been built for the development of big programs. Therefore the basic entity of APEX is a *subsystem*, a directory with certain traits recognized by APEX. All Ada compilation units have to reside in subsystems.

You can define an *export set*, i.e. the set of Ada units visible to other subsystems. However for a subsystem A to gain visibility to another subsystem B, A has to *import* B. After importing, A sees all units in B's export set. (This is much like the with-clauses, but here visibility means only potential visibility for Ada: units to be actually visible must be mentioned in a with-clause of course; units not in the export set cannot be used in with-clauses of Ada units in external subsystems.)

Normally subsystems should be hierarchically ordered, i.e. form a directed graph. But for special uses, subsystems can also mutually import one another

For configuration management, a subsystem is decomposed in *views*, subdirectories of the subsystem. Views hold different development versions of the Ada units. So actually it's not subsystems which import other subsystems, rather subsystem views import views of other subsystems. (Of course, the closure of all imports must be consistent - it cannot be the case that e.g. subsystem (A, view A1) imports subsystems (B, B1) and (C, C1), whereas (B, B1) imports (C, C2)).

A view can be defined to be the development view. Other views then hold releases at different stages.

Each Ada compilation unit has to reside in a file of its own. When compiling an Ada unit, the compiler follows the with-clauses. If a unit is not found within the subsystem holding the compile, the compiler searches the import list (only the direct imports are considered, not the closure).

Units can be taken under version control. In each subsystem, a set of *histories* can be defined. An Ada unit can be taken under control in a history. If you want to edit it, you first have to check it out - it gets a new version number. After the changes, you can check it in again, i.e. make the changes permanent (or you abandon your changes, i.e. go back to the previous version). You normally check out units in the

development view only; check-outs in release views can be forbidden.

You can select which version shall be the active one; normally it is the one latest checked in. You can even switch histories to get different development paths. e.g. different bodies of the same specification for different targets.

ObjectAda

ObjectAda is a set of tools for editing, compiling, navigating and debugging programs written in Ada. There are various editions of ObjectAda. With some editions you compile programs for the same platform and operating systems on which you run the tools. These are called native. With others, you can produce programs for different operating systems and platforms. One possible platform is the Java virtual machine.

These remarks apply to the native Microsoft Windows edition. You can run the translation tools either from the IDE or from the command line.

Whether you prefer to work from the IDE, or from the command line, a little bookkeeping is required. This is done by creating a project. Each project consists of a number of source files, and a number of settings like search paths for additional Ada libraries and other dependences. Each project also has at least one target. Typically, there is a debug target, and a release target. The names of the targets indicate their purpose. At one time you compile for debugging, typically during development, at other times you compile with different settings, for example when the program is ready for release. Some (all commercial?) editions of ObjectAda permit a Java (VM) target.

DEC Ada for VMS

DEC Ada is an Ada 83 compiler for VMS. While “DEC Ada” is probably the name most users know, the compiler is now called “HP Ada”. It had previously been known also by names of "VAX Ada" and "Compaq Ada".

DEC Ada uses a true library management system - so the first thing you need to do is create and activate a library:

```
ACS Library Create [MyLibrary]
ACS Set Library [MyLibrary]
```

When creating a library you already set some constraints like support for Long_Float or the available memory size. So carefully read

```
HELP ACS Library Create *
```

Then next step is to load your Ada sources into the library:

```
ACS Load [Source]*.ada
```

The sources don't need to be perfect at this stage but syntacticly correct enough for the compiler to determine the packages declared and analyze the **with** statements. Dec Ada allows you to have more then one package in one source file and you have any filename convention you like. The purpose of `ACS Load` is the creation of the dependency tree between the source files.

Next you compile them:

```
ACS Compile *
```

Note that compile take the package name and not the filename. The wildcard * means *all packages loaded*. The compiler automatically determines the right order for the compilation so a make tool is not strictly needed.

Last not least you link your file into an

```
ACS Link /Executable=[Executables]Main.exe Main
```

On large systems you might want to break sources down into several libraries - in which case you also need

```
ACS Merge /Keep *
```

to merge the content of the current library with the library higher up the hierarchy. The larger libraries should then be created with:

```
ACS Library Create /Large
```

This uses a different directory layout more suitable for large libraries.

External links:

- Developing Ada Products on OpenVMS (PDF)
- DEC Ada — Language Reference Manual (PDF)
- DEC Ada — Run-Time Reference (PDF)

DEC Ada IDE

Dec Ada comes without an IDE, however the Mode of the Vim text editor supports DEC Ada.

To be completed

You can help Wikibooks by adding the build information for other compilers. Click on the edit link over this title.

Compiling our Demo Source

Once you have downloaded our example programs you might wonder how to compilethem.

First you need to extract the sources. Use your favorite zip tool to achieve that. On extraction a directory with the same name as the filename is created. Beware: WinZip might also create a directory equaling the filename so Windows users need to be careful using the right option otherwise they end up with *wikibook-ada-1_2_0.src\wikibook-ada-1_2_0*.

Once you extracted the files you will find all sources in *wikibook-ada-1_2_0/Source*. You could compile them right there. For your convenience we also provide ready made project files for the following IDEs (If you find a directory for an IDEs not named it might be in the making and not actually work).

GNAT

You will find multi-target GNAT Project files and a multi-make Makefile file in *wikibook-ada-2_0_0/GNAT*. For i686 Linux and Windows you can compile any demo using:

```
gnat make -P project_file
```

You can also open them inside the GPS with

```
gps -P project_file
```

For other target platform it is a bit more difficult since you need to tell the project files which target you want to create. The following options can be used:

style ("Debug", "Release")
: you can define if you like a debug or release version so you can compare how the options affect size and speed.

os ("Linux", "OS2", "Windows_NT", "VMS")
: choose your operating system. Since there is no Ada 2005 available for OS/2 don't expect all examples to compile.

target ("i686", "x86_64", "AXP")
: choose your CPU - "i686" is any form of 32bit Intel or AMD CPU, "x86_64" is an 64 bit Intel or AMD CPU and if you have an "AXP" then you know it.

Remember to type all options as they are shown. To compile a debug version on x86_64 Linux you type:

```
gnat make -P project_file -Xstyle=Debug -Xos=Linux -Xtarget=x86_64
```

As said in the beginning there is also a **makefile** available that will automatically determine the target used. So if you have a GNU make you can save yourself a lot of typing by using:

```
make project
```

or even use

```
make all
```

to make all examples in debug and release in one go.

Each compile is stored inside it's own directory which is created in the form of *wikibook-ada-2_0_0/GNAT/**OS-Target-Style***. Empty directories are provided inside the archive.

Rational APEX

APEX uses the subsystem and view directory structure, so you will have to create those first and

copy the source files into the view. After creating a view using the architecture model of your choice, use the menu option "Compile -> Maintenance -> Import Text Files". In the Import Text Files dialog, add "wikibook-ada-2_0_0/Source/*.ad?" to select the Ada source files from the directory you originally extracted to. Apex uses the file extensions .1.ada for specs and .2.ada for bodies — don't worry, the import text files command will change these automatically.

To link an example, select its main subprogram in the directory viewer and click the link button in the toolbar, or "Compile -> Link" from the menu. Double-click the executable to run it. You can use the shift-key modifier to bypass the link or run dialog.

ObjectAda

ObjectAda commandline

The following describes using the ObjectAda tools for Windows in a console window.

Before you can use the ObjectAda tools from the command line, make sure the `PATH` environment variable lists the directory containing the ObjectAda tools. Something like

```
set path=%path%;P:\Programs\Aonix\ObjectAda\bin
```

A minimal ObjectAda project can have just one source file. like the Hello World program provided in

File: hello_world_1.adb (view, plain text, download page, browse all)

To build an executable from this source file, follow these steps (assuming the current directory is a fresh one and contains the above mentioned source file):

1. Register your source files.

```
X:\some\directory> adareg hello_world_1.adb
```

This makes your sources known to the ObjectAda tools. Have a look at the file UNIT.MAP created by adareg in the current directory if you like seeing what is happening under the hood.

1. Compile the source file.

```
X:\some\directory> adacomp hello_world_1.adb
Front end of hello_world_1.adb succeeded with no errors.
```

1. Build the executable program.

```
X:\some\directory> adabuild hello_world_1
ObjectAda Professional Edition Version 7.2.2: adabuild
   Copyright (c) 1997-2002 Aonix.  All rights reserved.
Linking...
Link of hello completed successfully
```

Notice that you specify the name of the main unit as argument to `adabuild`, not the name of the source file. In this case, it is *Hello_World_1* as in

```
procedure Hello_World_1 is
```

More information about the tools can be found in the user guide *Using the command line interface*, installed with the ObjectAda tools.

See also

- GNAT Online Documentation
  - GNAT User's Guide

5 CONTROL

Conditionals

Conditional clauses are blocks of code that will only execute if a particular expression (the condition) is true.

if-else

The if-else statement is the simplest of the conditional statements. They are also called branches, as when the program arrives at an "if" statement during its execution, control will "branch" off into one of two or more "directions". An if-else statement is generally in the following form:

```
if condition then
    statement;
else
    other statement;
end if;
```

If the original condition is met, then all the code within the first statement is executed. The optional else section specifies an alternative statement that will be executed if the condition is false. Exact syntax will vary between programming languages, but the majority of programming languages (especially procedural and structured languages) will have some form of if-else conditional statement built-in. The if-else statement can usually be extended to the following form:

```
if condition then
    statement;
elsif condition then
    other statement;
elsif condition then
    other statement;
...
else condition then
    another statement;
end if;
```

Only one statement in the entire block will be executed. This statement will be the first one with a condition which evaluates to be true. The concept of an if-else-if structure is easier to understand with the aid of an example:

```
with Ada.Text_IO;
use  Ada.Text_IO;
...
type Degrees is new Float range -273.15 .. Float'Last;
...
Temperature : Degrees;
...
if Temperature >= 40.0 then
    Put_Line ("It's extremely hot");
elsif Temperature >= 30.0 then
    Put_Line ("It's hot");
elsif Temperature >= 20.0 then
    Put_Line ("It's warm");
```

```
elsif Temperature >= 10.0 then
    Put_Line ("It's cool");
elsif Temperature >= 0.0 then
    Put_Line ("It's cold");
else
    Put_Line ("It's freezing");
end if;
```

Optimizing hints

When this program executes, the computer will check all conditions in order until one of them matches its concept of truth. As soon as this occurs, the program will execute the statement immediately following the condition and continue on, without checking any other condition for truth. For this reason, when you are trying to optimize a program, it is a good idea to sort your if-else conditions in descending probability. This will ensure that in the most common scenarios, the computer has to do less work, as it will most likely only have to check one or two "branches" before it finds the statement which it should execute. However, when writing programs for the first time, try not to think about this too much lest you find yourself undertaking premature optimization.

Having said all that, you should be aware that an optimizing compiler might rearrange your *if statement* at will when the statement in question is free from side effects. Among other techniques optimizing compilers might even apply jump tables and binary searches.

In Ada, conditional statements with more than one conditional do not use short-circuit evaluation by default. In order to mimic C/C++'s short-circuit evaluation, use **`and then`** or **`or else`** between the conditions.

case

Often it is necessary to compare one specific variable against several constant expressions. For this kind of conditional expression the case statement exists. For example:

```
case X is
   when 1 =>

      Walk_The_Dog;

   when 5 =>

      Launch_Nuke;

   when 8 | 10 =>

      Sell_All_Stock;

   when others =>

      Self_Destruct;

end case;
```

Unconditionals

Unconditionals let you change the flow of your program without a condition. You should be careful when using unconditionals. Often they make programs difficult to understand. Read Isn't goto evil? for more information.

return

End a function and return to the calling procedure or function.

For procedures:

```
return;
```

For functions:

```
return Value;
```

goto

Goto transfers contrd to the statement after the label.

```
   goto Label;
   Dont_Do_Something;
<<Label>>
   ...
```

Isn't goto evil?

One often hears that **goto** is *evil* and one should avoid using **goto**. But it is often overlooked that any return which is not the last statement inside a procedure or function is also an unconditional statement - a **goto** in disguise.

Therefore if you have functions and procedures with more than one **return** statement you can just as well use **goto**. When it comes down to readability the following two samples are almost the same:

```
procedure Use_Return is
begin
   Do_Something;

   if Test then
      return;
   end if;
   Do_Something_Else;
   return;
end Use_Return;
```

```
procedure Use_Goto is
begin
   Do_Something;

   if Test then
      goto Exit_Use_Goto;
   end if;

   Do_Something_Else;

<<Exit_Use_Goto>>
   return;
end Use_Goto;
```

Because the use of a **goto** needs the declaration of a label, the **goto** is in fact twice as readable than the use of **return**. So if readability is your concern and not a strict "don't use goto" programming rule then you should rather use **goto** than multiple **return**s. Best, of course, is the structured approach where neither **goto** nor multiple **returns** are needed:

```
procedure Use_If is
begin
   Do_Something;

   if not Test then

      Do_Something_Else;

   end if;
   return;
end Use_If;
```

Loops

Loops allow you to have a set of statements repeated over and over again.

Endless Loop

The endless loop is a loop which never ends and the statements inside are repeated forever. Never is meant as a relative term here — if the computer is switched off then even endless loops will end very abruptly.

```
Endless_Loop :
   loop

      Do_Something;

   end loop Endless_Loop;
```

The loop name (in this case, "Endless_Loop") is an optional feature of Ada. Naming loops is nice for readability but not strictly needed. Loop names are useful though if the program should jump out of an inner loop, see below.

Loop with Condition at the Beginning

This loop has a condition at the beginning. The statements are repeated as long as the condition is

met. If the condition is not met at the very beginning then the statements inside the loop are never executed.

```
While_Loop :
   while X <= 5 loop

      X := Calculate_Something;

   end loop While_Loop;
```

loop with condition at the end

This loop has a condition at the end and the statements are repeated until the condition is met. Since the check is at the end the statements areat least executed once.

```
Until_Loop :
   loop

      X := Calculate_Something;

      exit Until_Loop when X > 5;
   end loop Until_Loop;
```

loop with condition in the middle

Sometimes you need to first make a calculation and exit the loop when a certain criterion is met. However when the criterion is not met there is something else to be done. Hence you need a loop where the exit condition is in the middle.

```
Exit_Loop :
   loop

      X := Calculate_Something;

      exit Exit_Loop when X > 5;
      Do_Something (X);
   end loop  Exit_Loop;
```

In Ada the **exit** condition can be combined with any other loop statement as well. You can also have more then one **exit** statement. You can also exit a named outer loop if you have several loops inside each other.

for loop

Quite often one needs a loop where a specific variable is counted from a given start value up or down to a specific end value. You could use the while loop here - but since this is a very common loop there is an easier syntax available.

```
For_Loop :
   for I in Integer range 1 .. 10 loop

      Do_Something (I)

   end loop For_Loop;
```

You don't have to declare both type and range as seen in the example. If you leave out the type then the compiler will determine the type by context and leave out the range then the loop will iterate over every valid value for the type given.

As always with Ada: when "determine by context" gives two or more possible options then an error will be displayed and then you have to name the type to be used. Ada will only do "guess-works" when it is safe to do so.

for loop on arrays

Another very common situation is the need for a loop which iterates over every element of an array. The following sample code shows you how to achieve this:

```
Array_Loop :
   for I in X'Range loop

      X (I) := Get_Next_Element;

   end loop Array_Loop;
```

With X being an array. Note: This syntax is mostly used on arrays - hence the name - but will also work with other types when a full iteration is needed.

Working Demo

The following Demo shows how to iterate over every element of an integer type.

File: range_1.adb (view, plain text, download page, browse all)

```
with Ada.Text_IO;
procedure Range_1 is
   type Range_Type is range -5 .. 10;
   package T_IO renames Ada.Text_IO;
   package I_IO is new  Ada.Text_IO.Integer_IO (Range_Type);
begin
   for A in Range_Type loop
      I_IO.Put (Item  => A,
                Width => 3,
                Base  => 10);

      if A < Range_Type'Last then
         T_IO.Put (",");
      else
         T_IO.New_Line;
      end if;
   end loop;
end Range_1;
```

See also

Wikibook

- Ada Programming

Ada Reference Manual

- 5.3 If Statements (Annotated)
- 5.4 Case Statements (Annotated)
- 5.5 Loop Statements (Annotated)
- 5.6 Block Statements (Annotated)
- 5.7 Exit Statements (Annotated)
- 5.8 Goto Statements (Annotated)
- 6.5 Return Statements (Annotated)

6 Type System

Ada's type system allows the programmer to construct powerful abstractions that represent the real world, and to provide valuable information to the compiler, so that the compiler can find many logic or design errors before they become bugs. It is at the heart of the language, and good Ada programmers learn to use it to great advantage. Four principles govern the type system:

- **Strong typing**: types are incompatible with one another, so it is not possible to mix apples and oranges. There are, however, ways to convert between types.
- **Static typing**: the programmer must declare the type of all objects, so that the compiler can check them. This is in contrast to dynamically typed languages such as Python, Smalltalk, or Lisp. Also, there is no type inference like in OCaml.
- **Abstraction**: types represent the real world or the problem at hand; not how the computer represents the data internally. There are ways to specify exactly how a type must be represented at the bit level, but we will defer that discussion to another chapter.
- **Name equivalence**, as opposed to *structural equivalence* used in most other languages. Two types are compatible if and only if they have the same name; *not* if they just happen to have the same size or bit representation. You can thus declare two integer types with the same ranges that are totally incompatible, or two record types with the exact same components, but which are incompatible.

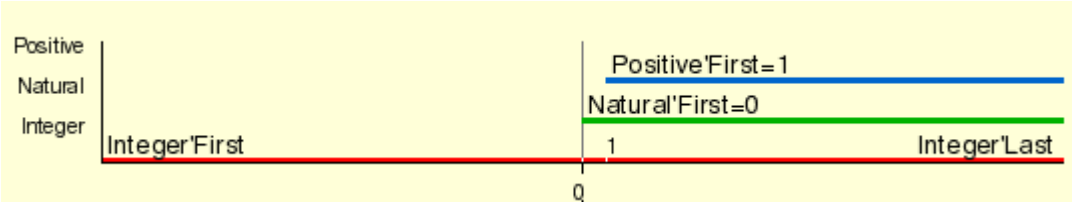
Types are incompatible with one another. However, each type can have any number of *subtypes*, which are compatible with one another, and with their base type.

Predefined types

There are several predefined types, but most programmers prefer to define their own, application-specific types. Nevertheless, these predefined types are very useful as interfaces between libraries developed independently. The predefined library, obviously, uses these types too.

These types are predefined in the Standard package:

Integer
: This type covers at least the range -2**15+1 .. +2**15-1 (RM 3.5.4 (21) (Annotated)). The Standard also defines Natural and Positive subtypes of this type.



Float
: There is only a very weak implementation requirement on this type (RM 3.5.7 (14) (Annotated)); most of the time you would define your own floating-point types, and specify your precision and range requirements.

Duration
: A fixed point type used for timing. It represents a period of time in seconds (RM A.1 (43) (Annotated)).

Character
: A special form of Enumerations. There are two predefined kinds of character types: 8-bit characters (called Character) and 16-bit characters (called Wide_Character). Ada 2005 adds a 32-bit character type called Wide_Wide_Character.

String and Wide_String
: Two indefinite array types, of Character and Wide_Character respectively. Ada 2005 adds a Wide_Wide_String type. The standard library contains packages for handling strings in three variants: fixed length (Ada.Strings.Fixed), with varying length below a certain upper bound (Ada.Strings.Bounded), and unbounded length (Ada.Strings.Unbounded). Each of these packages has a Wide_ and a Wide_Wide_ variant.

Boolean
: A Boolean in Ada is an Enumeration of False and True with special semantics.

Packages System and System.Storage_Elements predefine some types which are primarily useful for low-level programming and interfacing to hardware.

System.Address
: An address in memory.

System.Storage_Elements.Storage_Offset
: An offset, which can be added to an address to obtain a new address. You can also substract one address from another to get the offset between them. Together, Address, Storage_Offset and their associated subprograms provide for address arithmetic.

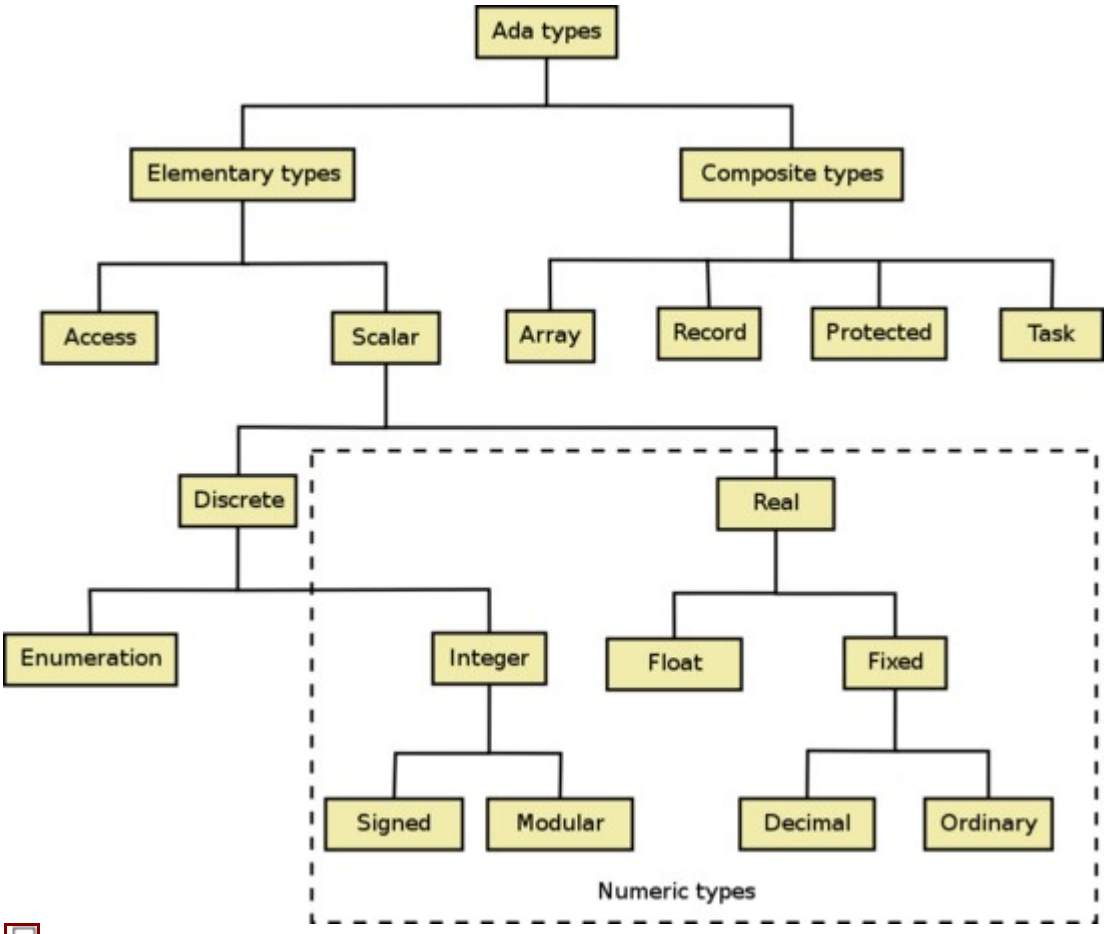
System.Storage_Elements.Storage_Count
: A subtype of Storage_Offset which cannot be negative, and represents the memory size of a data structure (similar to C's `size_t`).

System.Storage_Elements.Storage_Element
: In most computers, this is a byte. Formally, it is the smallest unit of memory that has an address.

System.Storage_Elements.Storage_Array
: An array of Storage_Elements without any meaning, useful when doing raw memory access.

The Type Hierarchy

Types are organized hierarchically. A type inherits properties from types above it in the hierarchy. For example, all scalar types (integer, enumeration, modular, fixed-point and floating-point types) have operators "<", ">" and arithmetic operators defined for them, and all discrete types can serve as array indexes.



Ada type hierarchy

Here is a broad overview of each category of types; please follow the links for detailed explanations. Inside parenthesis there are equivalences in C and Pascal for readers familiar with those languages.

Signed Integers (**int**, **INTEGER**)
: Signed Integers are defined via the range of values needed.

Unsigned Integers (**unsigned**, **CARDINAL**)
: Unsigned Integers are called Modular Types. Apart from being unsigned they also have wrap-around functionality.

Enumerations (**enum**, **char**, **bool**, **BOOLEAN**)
: Ada Enumeration types are a separate type family.

Floating point (**float**, **double**, **REAL**)
: Floating point types are defined by the digits needed, the relative error bound.

Ordinary and Decimal Fixed Point (**DECIMAL**)
: Fixed point types are defined by their delta, the absolute error bound.

Arrays (**[ ]**, **ARRAY [ ] OF**, **STRING**)
: Arrays with both compile-time and run-time determined size are supported.

Record (**struct**, **class**, **RECORD OF**)
: A **record** is a composite type that groups one or more fields.

Access (*, ^, **POINTER TO**)
: Ada's Access types may be more than just a simple memory address.

Task & Protected (no equivalence in C or Pascal)
: Task and Protected types allow the control of concurrency

Interfaces (no equivalence in C or Pascal)

New in Ada 2005, these types are similar to the Java interfaces.

Concurrency Types

The Ada language uses types for one more purpose in addition to classifying data + operations. The type system integrates concurrency (threading, parallelism). Programmers will use types for expressing the concurrent threads of control of their programs.

The core pieces of this part of the type system, the **task** types and the **protected** types are explained in greater depth in a section on tasking.

Limited Types

Limiting a type means disallowing assignment. The "concurrency types" described above are always limited. Programmers can define their own types to be limited, too, like this:

```
type T is limited …;
```

(The ellipsis stands for **private**, or for a **record** definition, see the corresponding subsection on this page.) A limited type also doesn't have an equality operator unless the programmer defines one.

You can learn more in the limited types chapter.

Defining new types and subtypes

You can define a new type with the following syntax:

```
type T is...
```

followed by the description of the type, as explained in detail in each category of type.

Formally, the above declaration creates a type and its first *subtype* named `T`. The type itself is named `T'Base`. But this is an academic consideration; for most purposes, it is sufficient to think of `T` as a type.

As explained above, all types are incompatible; thus:

```
type Integer_1 is range 1 .. 10;
type Integer_2 is range 1 .. 10;
A : Integer_1 := 8;
B : Integer_2 := A; -- illegal!
```

is illegal, because `Integer_1` and `Integer_2` are different and incompatible types. It is this feature which allows the compiler to detect logic errors at compile time, such as adding a file descriptor to a number of bytes, or a length to a weight. The fact that the two types have the same range does not make them compatible: this is *name equivalence* in action, as opposed to structural equivalence. (Below, we will see how you can convert between incompatible types; there are strict rules for this.)

Creating subtypes

You can also create new subtypes of a given type, which will be compatible with each other, like this:

```
type Integer_1 is range 1 .. 10;
subtype Integer_2 is Integer_1 range 7 .. 11;
A : Integer_1 := 8;
B : Integer_2 := A; -- OK
```

Now, `Integer_1` and `Integer_2` are compatible because they are both subtypes of the same type, namely `Integer_1'Base`.

It is not necessary that the ranges overlap, or be included in one another. The compiler inserts a run-time range check when you assign A to B; if the value of A, at that point, happens to be outside the range of `Integer_2`, the program raises `Constraint_Error`.

There are a few predefined subtypes which are very useful:

```
subtype Natural  is Integer range 0 .. Integer'Last;
subtype Positive is Integer range 1 .. Integer'Last;
```

Derived types

A derived type is a new, full-blown type created from an existing one. Like any other type, it is incompatible with its parent; however, it inherits the primitive operations defined for the parent type.

```
type Integer_1 is range 1 .. 10;
type Integer_2 is new Integer_1 range 2 .. 8;
A : Integer_1 := 8;
B : Integer_2 := A; -- illegal!
```

Here both types are discrete; it is mandatory that the range of the derived type be included in the range of its parent. Contrast this with subtypes. The reason is that the derived type inherits the primitive operations defined for its parent, and these operations assume the range of the parent type. Here is an illustration of this feature:

```
procedure Derived_Types is
   package Pak is
      type Integer_1 is range 1 .. 10;
      procedure P (I : in Integer_1); -- primitive operation, assumes 1 .. 10
      type Integer_2 is new Integer_1 range 8 .. 10; -- must not break P's assumption
   end Pak;
   package body Pak is
      -- omitted
   end Pak;
   use Pak;
   A : Integer_1 := 4;
   B : Integer_2 := 9;
begin
   P (B); -- OK, call the inherited operation
end Derived_Types;
```

When we call `P (B)`, there is *no* type conversion; we call a completely new version of `P`, which accepts only parameters of type `Integer_2`. Since there is no type conversion, there is no range check either. This is why the set of acceptable values for the derived type (here, 8 .. 10) must be included in that of the parent type (1 .. 10).

Subtype categories

Ada supports various categories of subtypes which have different abilities. Here is an overview in alphabetical order.

Anonymous subtype

A subtype which does not have a name assigned to it. Such a subtype is created with a variable declaration:

```
X : String (1 .. 10) := (others => ' ');
```

Here, (1 .. 10) is the constraint. This variable declaration is equivalent to:

```
subtype Anonymous_String_Type is String (1 .. 10);
X : Anonymous_String_Type := (others => ' ');
```

Base type

Within this document we almost always speak of subtypes. But where there is a subtype there must also be a base type from which it is derived. In Ada all base types are anonymous and only subtypes may be named. However, it is still possible to use base type by the use of the 'Base attribute.

Constrained subtype

A subtype of an indefinite subtype that adds a constraint. The following example defines a 10 character string sub-type.

```
subtype String_10 is String (1 .. 10);
```

If all constraints of an original indefinite subtype are defined then the new sub-type is a definite subtype.

Definite subtype

A definite subtype is a subtype whose size is known at compile-time. All subtypes which are not indefinite subtypes are, by definition, definite subtypes.

Objects of definite subtypes may be declared without additional constraints.

Indefinite subtype

An **indefinite subtype** is a subtype whose size is not known at compile-time but is dynamically calculated at run-time. An indefinite subtype does not by itself provide enough information to create an object; an additional constraint or explicit initialization expression is necessary in order to calculate the actual size and therefore create the object.

```
X : String := "This is a string";
```

X is an object of the indefinite (sub)type String. Its constraint is derived implicitly from its initial value. X may change its value, but not its bounds.

It should be noted that it is not necessary to initialize the object from a literal. You can also use a function. For example:

```
X : String := Ada.Command_Line.Argument (1);
```

This statement reads the first command-line argument and assigns it to X.

Named subtype

A subtype which has a name assigned to it. These subtypes are created with the keyword **type** (remember that types are always anonymous, the name in a type declaration is the name of the first subtype) or **subtype**. For example:

```
type Count_To_Ten is range 1 .. 10;
```

Count_to_Ten is the first subtype of a suitable integer base type. If you however would like to use this as an index constraint on String, the following declaration is illegal:

```
subtype Ten_Characters is String (Count_to_Ten);
```

This is because String has Positive as index, which is a subtype of Integer (these declarations are taken from package Standard):

```
subtype Positive is Integer range 1 .. Integer'Last;
type String is (Positive range <>) of Character;
```

So you have to use the following declarations:

```
subtype Count_To_Ten is Integer range 1 .. 10;
subtype Ten_Characters is String (Count_to_Ten);
```

Now Ten_Characters is the name of that subtype of String which is constrained to Count_To_Ten. You see that posing constraints on types versus subtypes has very different effects.

Unconstrained subtype

A subtype of an indefinite subtype that does not add a constraint only introduces a new name for the original subtype.

```
subtype My_String is String;
```

`My_String` and `String` are interchangeable.

Qualified expressions

In most cases, the compiler is able to infer the type of an expression; for example:

```
type Enum is (A, B, C);
E : Enum := A;
```

Here the compiler knows that A is a value of the type Enum. But consider:

```
procedure Bad is
   type Enum_1 is (A, B, C);
   procedure P (E : in Enum_1) is... -- omitted
   type Enum_2 is (A, X, Y, Z);
   procedure P (E : in Enum_2) is... -- omitted
begin
   P (A); -- illegal: ambiguous
end Bad;
```

The compiler cannot choose between the two versions of P; both would be equally valid. To remove the ambiguity, you use a *qualified expression*:

```
   P (Enum_1'(A)); -- OK
```

As seen in the following example, this syntax if often used when creating new objects. If you try to compile the example, it will fail with a compilation error since the compiler will determine that 256 is not in range of Byte.

File: convert_evaluate_as.adb (view, plain text, download page, browse all)

```
with Ada.Text_IO;
procedure Convert_Evaluate_As is
   type Byte     is mod 2**8;
   type Byte_Ptr is access Byte;
   package T_IO renames Ada.Text_IO;
   package M_IO is new Ada.Text_IO.Modular_IO (Byte);
   A : constant Byte_Ptr := new Byte'(256);
begin
   T_IO.Put ("A = ");
   M_IO.Put (Item  => A.all,
             Width =>  5,
             Base  => 10);
end Convert_Evaluate_As;
```

Type conversions

Data does not always come in the format you need them and you face the task of converting them. Ada as a true multi-purpose language with a special emphasis on "mission critical", "system programming" and "safety" has several conversion techniques. As the most difficult part is choosing the right one, I have sorted them so that you should try the first one first; the last technique is "the last resort if all other fails". Also added are a few related techniques which you might choose instead of actually converting the data.

Since the most important aspect is not the result of a successful conversion, but how the system will react to an invalid conversion, all examplesalso demonstrate **faulty** conversions.

Explicit type conversion

An explicit type conversion looks much like a function call; it does not use the *tick* (apostrophe, ') like the qualified expression does.

```
Type_Name (Expression)
```

The compiler first checks that the conversion is legal, and if it is, it inserts a run-time check at the point of the conversion; hence the name *checked conversion*. If the conversion fails, the program raises Constraint_Error. Most compilers are very smart and optimise away the constraint checks; so, you need not worry about any performance penalty. Some compilers can also warn that a constraint check will always fail (and optimise the check with an unconditional raise).

Explicit type conversions are legal:

- between any two numeric types
- between any two subtypes of the same type
- between any two types derived from the same type
- and *nowhere else*

```
I: Integer := Integer (10);   -- Unnecessary explicit type conversion
J: Integer := 10;             -- Implicit conversion from universal integer
K: Integer := Integer'(10);   -- Use the value 10 of type Integer: qualified expression
                              -- (qualification not necessary here).
```

This example illustrates explicit type conversions:

File: convert_checked.adb (view, plain text, download page, browse all)

```
with Ada.Text_IO;
procedure Convert_Checked is
   type Short is range -128 .. +127;
   type Byte  is mod 256;
   package T_IO renames Ada.Text_IO;
   package I_IO is new Ada.Text_IO.Integer_IO (Short);
   package M_IO is new Ada.Text_IO.Modular_IO (Byte);
   A : Short := -1;
   B : Byte;
begin
   B := Byte (A);
   T_IO.Put ("A = ");
   I_IO.Put (Item  =>  A,
             Width =>  5,
             Base  => 10);
   T_IO.Put (", B = ");
   M_IO.Put (Item  =>  B,
             Width =>  5,
             Base  => 10);
end Convert_Checked;
```

Explicit conversions are possible between any two numeric types: integers, fixed-point and floating-point types. If one of the types involved is a fixed-point or floating-point type, the compiler not only checks for the range constraints, but also performs any loss of precision necessary.

Example 1: the loss of precision causes the procedure to only ever print "0" or "1", since `P / 100` is an integer and is always zero or one.

```
with Ada.Text_IO;
procedure Naive_Explicit_Conversion is
   type Proportion is digits 4 range 0.0 .. 1.0;
   type Percentage is range 0 .. 100;
   function To_Proportion (P : in Percentage) return Proportion is
```

```
   begin
      return Proportion (P / 100);
   end To_Proportion;
begin
   Ada.Text_IO.Put_Line (Proportion'Image (To_Proportion (27)));
end Naive_Explicit_Conversion;
```

Example 2: we use an intermediate floating-point type to guarantee the precision.

```
with Ada.Text_IO;
procedure Explicit_Conversion is
   type Proportion is digits 4 range 0.0 .. 1.0;
   type Percentage is range 0 .. 100;
   function To_Proportion (P : in Percentage) return Proportion is
      type Prop is digits 4 range 0.0 .. 100.0;
   begin
      return Proportion (Prop (P) / 100.0);
   end To_Proportion;
begin
   Ada.Text_IO.Put_Line (Proportion'Image (To_Proportion (27)));
end Explicit_Conversion;
```

Change of Representation

Type conversions can be used for packing and unpacking of records or arrays.

```
type Unpacked is record
  -- any components
end record;
type Packed is new Unpacked;
for  Packed use record
  -- component clauses for some or for all components
end record;


P: Packed;
U: Unpacked;
P := Packed (U);   -- packs U
U := Unpacked (P);   -- unpacks P
```

Checked conversion for non-numeric types

The examples above all revolved around conversions between numeric types; it is possible to convert between any two numeric types in this way. But what happens between non-numeric types, e.g. between array types or record types? The answer is two-fold:

- you can convert explicitly between a type and types derived from it, or between types derived from the same type,
- and that's all. No other conversions are possible.

Why would you want to derive a record type from another record type? Because of representation clauses. Here we enter the realm of low-level systems programming, which is not for the faint of heart, nor is it useful for desktop applications. So hold on tight, and let's dive in.

Suppose you have a record type which uses the default, efficient representation. Now you want to write this record to a device, which uses a special record format. This special representation is more compact (uses fewer bits), but is grossly inefficient. You want to have a layered programming interface: the upper layer, intended for applications, uses the efficient representation. The lower layer is a device driver that accesses the hardware directly and uses the inefficient representation.

```
package Device_Driver is
```

```
   type Size_Type is range 0 .. 64;
   type Register is record
      A, B : Boolean;
      Size : Size_Type;
   end record;
   procedure Read (R : out Register);
   procedure Write (R : in Register);
end Device_Driver;
```

The compiler chooses a default, efficient representation for `Register`. For example, on a 32-bit machine, it would probably use three 32-bit words, one for A, one for B and one for Size. This efficient representation is good for applications, but at one point we want to convert the entire record to just 8 bits, because that's what our hardware requires.

```
package body Device_Driver is
   type Hardware_Register is new Register; -- Derived type.
   for Hardware_Register use record
      A at 0 range 0 .. 0;
      B at 0 range 1 .. 1;
      Size at 0 range 2 .. 7;
   end record;
   function Get return Hardware_Register; -- Body omitted
   procedure Put (H : in Hardware_Register); -- Body omitted

   procedure Read (R : out Register) is
      H : Hardware_Register := Get;
   begin
      R := Register (H); -- Explicit conversion.
   end Read;
   procedure Write (R : in Register) is
   begin
      Put (Hardware_Register (R)); -- Explicit conversion.
   end Write;
end Device_Driver;
```

In the above example, the package body declares a derived type with the inefficient, but compact representation, and converts to and from it.

This illustrates that **type conversions can result in a change of representation**.

View conversion, in object-oriented programming

Within object-oriented programming you have to distinguish between specific types and class-wide types.

With specific types, only conversions to ancestors are possible and, of course, are checked. During the conversion, you do not "drop" any components that are present in the derived type and not in the parent type; these components are still present, you just don't see them anymore. This is called a *view conversion*.

There are no conversions to derived types (where would you get the further components from?); *extension aggregates* have to be used instead.

```
type Parent_Type is tagged null record;
type Child_Type  is new Parent_Type with null record;
Child_Instance  : Child_Type;
-- View conversion from the child type to the parent type:
Parent_View : Parent_Type := Parent_Type (Child_Instance);
```

Since, in object-oriented programming, an object of child type *is an* object of the parent type, no run-time check is necessary.

With class-wide types, conversions to ancestor and child types are possible and are checked as well. These conversions are also view conversions, no data is created or lost.

```
procedure P (Parent_View : Parent_Type'Class) is
   -- View conversion to the child type:
   One : Child_Type := Child_Type (Parent_View);

   -- View conversion to the class-wide child type:
   Two : Child_Type'Class := Child_Type'Class (Parent_View);
```

This view conversion involves a run-time check to see if `Parent_View` is indeed a view of an object of type `Child_Type`. In the second case, the run-time check accepts objects of type `Child_Type` but also any type derived from `Child_Type`.

View renaming

A renaming declaration does not create any new object and performs no conversion; it only gives a new name to something that already exists. Performance is optimal since the renaming is completely done at compile time. We mention it here because it is a common idiom in object oriented programming to rename the result of a view conversion.

```
type Parent_Type is tagged null record;
type Child_Type  is new Parent_Type with null record;
Child_Instance : Child_Type;
Parent_View    : Parent_Type'Class renames Parent_Type'Class (Child_Instance);
```

Now, `Parent_View` is not a new object, but another name for `Child_Instance`.

Address conversion

Ada's access type is not just a memory location (a thin pointer). Depending on implementation and the access type used, the access might keep additional information (a fat pointer). For example GNAT keeps two memory addresses for each access to an indefinite object — one for the data and one for the constraint informations (Size, First, Last).

If you want to convert an access to a simple memory location you can use the package System.Address_To_Access_Conversions. Note however that an address and a fat pointer cannot be converted reversibly into one another.

The address of an array object is the address of its first component. Thus the bounds get lost in such a conversion.

```
type My_Array is array (Positive range <>) of Something;
A: My_Array (50 .. 100);
     A'Address = A(A'First)'Address
```

Unchecked conversion

One of the great criticisms of Pascal was "there is no escape". The reason was that sometimes you have to convert the incompatible. For this purpose, Ada has the generic function *Unchecked_Conversion*:

```
generic
   type Source (<>) is limited private;
```

```
   type Target (<>) is limited private;
function Ada.Unchecked_Conversion (S : Source) return Target;
```

Unchecked_Conversion will bit-copy the data and there are absolutely no checks. It is **your** chore to make sure that the requirements on unchecked conversion as stated in RM 13.9 (Annotated) are fulfilled; if not, the result is implementation dependent and may even lead to abnormal data.

A function call to (an instance of) *Unchecked_Conversion* will copy the source to the destination. The compiler may also do a conversion *in place* (every instance has the convention *Intrinsic*).

To use *Unchecked_Conversion* you need to instantiate the generic.

In the example below, you can see how this is done. When run, the example it will output "A = -1, B = 255". No error will be reported, but is this the result you expect?

File: convert_unchecked.adb (view, plain text, download page, browse all)

```
with Ada.Text_IO;
with Ada.Unchecked_Conversion;
procedure Convert_Unchecked is

   type Short is range -128 .. +127;
   type Byte  is mod 256;
   package T_IO renames Ada.Text_IO;
   package I_IO is new Ada.Text_IO.Integer_IO (Short);
   package M_IO is new Ada.Text_IO.Modular_IO (Byte);
   function Convert is new Ada.Unchecked_Conversion (Source => Short,
                                                     Target => Byte);
   A : constant Short := -1;
   B : Byte;
begin

   B := Convert (A);
   T_IO.Put ("A = ");
   I_IO.Put (Item  =>  A,
             Width =>  5,
             Base  => 10);
   T_IO.Put (", B = ");
   M_IO.Put (Item  =>  B,
             Width =>  5,
             Base  => 10);

end Convert_Unchecked;
```

Overlays

If the copying of the result of *Unchecked_Conversion* is too much waste in terms of performance, then you can try overlays, i.e. address mappings. By using overlays, both objects share the same memory location. If you assign a value to one, the other changes as well. The syntax is:

```
for Target'Address use expression;
pragma Import (Ada, Target);
```

where *expression* defines the address of the source object.

While overlays might look more elegant than Unchecked_Conversion, you should be aware that they are even more dangerous and have even greater potential for doing something very wrong. For example if `Source'Size < Target'Size` and you assign a value to Target, you might inadvertently write into memory allocated to a different object.

You have to take care also of implicit initializations of objects of the target type, since they would

overwrite the actual value of the source object. The Import pragma with convention Ada can be used to prevent this, since it avoids the implicit initialization, RM B.1 (Annotated).

The example below does the sameas the example from "Unchecked Conversion".

File: convert_address_mapping.adb (view, plain text, download page, browse all)

```
with Ada.Text_IO;
procedure Convert_Address_Mapping is
   type Short is range -128 .. +127;
   type Byte  is mod 256;
   package T_IO renames Ada.Text_IO;
   package I_IO is new Ada.Text_IO.Integer_IO (Short);
   package M_IO is new Ada.Text_IO.Modular_IO (Byte);
   A : aliased Short;
   B : aliased Byte;

   for B'Address use A'Address;
  pragma Import (Ada, B);

begin
   A := -1;
   T_IO.Put ("A = ");
   I_IO.Put (Item  =>  A,
             Width =>  5,
             Base  => 10);
   T_IO.Put (", B = ");
   M_IO.Put (Item  =>  B,
             Width =>  5,
             Base  => 10);
end Convert_Address_Mapping;
```

Export / Import

Just for the record: There is still another method using the `Export` and `Import` pragmas. However, since this method completely undermines Ada's visibility and type concepts even more than overlays, it has no place here in this language introduction and is left to experts.

See also

Wikibook

- Ada Programming

Ada Reference Manual

- 3.2.1 Type Declarations (Annotated)
- 3.3 Objects and Named Numbers (Annotated)
- 3.7 Discriminants (Annotated)
- 3.10 Access Types (Annotated)
- 4.9 Static Expressions and Static Subtypes (Annotated)
- 13.9 Unchecked Type Conversions (Annotated)
- 13.3 Operational and Representation Attributes (Annotated)
- Annex K (informative) Language-Defined Attributes (Annotated)

7 Integer types

A **range** is an integer value which rangesfrom a First to a last Last. It is defined as

```
range First .. Last
```

When a value is assigned to a range it is checked for vality and an exceptions is raised when the value is not within First to Last.

Working demo

The following Demo defines a new range from -5 to 10 and then prints the whole range out.

File: range_1.adb (view, plain text, download page, browse all)

```
with Ada.Text_IO;
procedure Range_1 is
   type Range_Type is range -5 .. 10;
   package T_IO renames Ada.Text_IO;
   package I_IO is new Ada.Text_IO.Integer_IO (Range_Type);
begin
   for A in Range_Type loop
      I_IO.Put (
         Item  => A,
         Width => 3,
         Base  => 10);

      if A < Range_Type'Last then
         T_IO.Put (",");
      else
         T_IO.New_Line;
      end if;
   end loop;
end Range_1;
```

See also

Wikibook

- Ada Programming
- Ada Programming/Types
- Ada Programming/Keywords/range

Ada Reference Manual

- 4.4 Expressions (Annotated)
- 3.5.4 Integer Types (Annotated)

8 Unsigned integer types

Description

Unsigned integers in Ada have a value range from 0 to some positive number (not necessarily one less than a power of 2). They are defined using the **mod** keyword because they implement a wrap-around arithmetic.

```
mod Modulus
```

where 'First is 0 and 'Last is Modulus - 1.

Wrap-around arithmetic means that 'Last + 1 = 0 = 'First, and 'First - 1 = 'Last. Additionally to the normal arithmetic operators, bitwise **and**, **or** and **xor** are defined for the type.

The predefined package Interfaces (RM B.2 (Annotated)) presents unsigned integers based on powers of 2

```
type Unsigned_n is mod 2**n;
```

for which also shift and rotate operations are defined. The values of *n* depend on compiler and target architecture.

You can use **range** to sub-range a modular type:

```
type Byte is mod 256;
subtype Half_Byte is Byte range 0 .. 127;
```

But beware: the Modulus of Half_Byte is still 256! Arithmetic with such a type is interesting to say the least.

See also

Wikibook

- Ada Programming
- Ada Programming/Types
- Ada Programming/Keywords/mod

Ada Reference Manual

- 4.4 Expressions (Annotated)
- 3.5.4 Integer Types (Annotated)

9 Enumerations

An **enumeration** type is defined as a list of possible values:

```
type Primary_Color is (Red, Green, Blue);
```

Like for numeric types, where e.g. 1 is an integer literal, Red, Green and Blue are called the literals of this type. There are no other values assignable to objects of this type.

Operators and attributes

Apart from equality ("="), the only operators on enumeration types are the ordering operators: "<", "<=", "=", "/=", ">=", ">", where the order relation is given implicitly by the sequence of literals: Each literal has a position, starting with 0 for the first, incremented by one for each successor. This position can be queried via the 'Pos attribute; the inverse is 'Val, which returns the corresponding literal. In our example:

```
Primary_Color'Pos (Red) = 0
Primary_Color'Val (0)   = Red
```

There are two other important attributes: Image and Value (don't confuse Val with Value). Image returns the string representation of the value (in capital letters), Value is the inverse:

```
Primary_Color'Image ( Red ) = "RED"
Primary_Color'Value ("Red") =  Red
```

These attributes are important for simple IO (there are more elaborate IO facilities in Ada.Text_IO for enumeration types). Note that, since Ada is case-insensitive, the string given to 'Value can be in any case.

Enumeration literals

Literals are overloadable, i.e. you can have another type with the same literals.

```
type Traffic_Light is (Red, Yellow, Green);
```

Overload resolution within the context of use of a literal normally resolves which Red is meant. Only if you have an unresolvable overloading conflict, you can qualify with special syntax which Red is meant:

```
Primary_Color'(Red)
```

Like many other declarative items, enumeration literals can be renamed. In fact, such a literal is a actually function, so it has to be renamed as such:

```
function Red return P.Primary_Color renames P.Red;
```

Here, Primary_Color is assumed to be defined in package P, which is visible at the place of the renaming declaration. Renaming makes Red directly visible without necessity to resort the use-clause.

Note that redeclaration as a function does not affect the staticness of the literal.

Characters as enumeration literals

Rather unique to Ada is the use of character literals as enumeration literals:

```
type ABC is ('A', 'B', 'C');
```

This literal 'A' has **nothing** in common with the literal 'A' of the predefined type Character (or Wide_Character).

Every type that has at least one character literal is a character type. For every character type, string literals and the concatenation operator "&" are also implicitly defined.

```
type My_Character is (No_Character, 'a', Literal, 'z');
type My_String is array (Positive range <>) of My_Character;
S: My_String := "aa" & Literal & "za" & 'z';
T: My_String := ('a', 'a', Literal, 'z', 'a', 'z');
```

In this example, S and T have the same value.

Ada's Character type is defined that way. See Ada Programming/Libraries/Standard.

Booleans as enumeration literals

Also Booleans are defined as eunumeration types

```
type Boolean is (False, True);
```

There is special semantics implied with this declaration in that objects and expressions of this type can be used as conditions. Note that the literals False and True are not Ada keywords.

Thus it is not sufficient to declare a type with these literals and then hope objects of this type can be used like so:

```
type My_Boolean is (False, True);
Condition: My_Boolean;
if Condition then  -- wrong, won't compile
```

If you need your own Booleans(perhaps with special size requirements), you have to derive from the predefined Boolean:

```
type My_Boolean is new Boolean;
Condition: My_Boolean;
if Condition then  -- OK
```

Enumeration subtypes

You can use `range` to subtype an enumeration type:

```
subtype Capital_Letter is Character range 'A' .. 'Z';
```

See also

Wikibook

- Ada Programming
- Ada Programming/Types
- Ada Programming/Libraries/Standard

Ada Reference Manual

- 3.5.1 Enumeration Types (Annotated)

10 Floating point types

Description

Wikipedia has related information at ***Floating point***.

To define a floating point type, you only have to say how many **digits** are needed, i.e. you define the relative precision:

```
digits Num_Digits
```

If you like, you can declare the minimum range needed as well:

```
digits Num_Digits range Low .. High
```

This facility is a great benefit of Ada over (most) other programming languages. In other languages, you just choose between "float" and "long float", and what most people do is

- choose float if they don't care about accuracy
- otherwise, choose long float, because it is the best you can get

In either case, you don't know what accuracy you get.

In Ada, you specify the accuracy you need, and the compiler will choose an appropriate floating point type with *at least* the accuracy you asked for. This way, your requirement is guaranteed. Moreover, if the computer has more than two floating point types available, the compiler can make use of all of them.

See also

Wikibook

- Ada Programming
- Ada Programming/Types
- Ada Programming/Types/range
- Ada Programming/Types/delta
- Ada Programming/Types/mod
- Ada Programming/Keywords/digits

Ada Reference Manual

- 3.5.7 Floating Point Types (Annotated)

FIXED POINT TYPES

Description

Wikipedia has related information at ***Fixed-point arithmetic***.

A fixed point type defines a set of values that are evenly spaced with a given absolute precision. In contrast, floating point values are all spaced according to a relative precision.

The absolute precision is given as the delta of the type. There are two kinds of fixed point types, ordinary and decimal.

For **Ordinary Fixed Point** types, the delta gives a hint to the compiler how to choose the small value if it is not specified: It can be *any power of two* not greater than delta. You may specify the small via an attribute clause to be *any value* not greater than delta. (If the compiler cannot conform to this small value, it has to reject the declaration.)

For **Decimal Fixed Point** types, the small is defined to be the delta, which in turn must be a power of ten. (Thus you cannot specify the small by an attribute clause.)

For example, if you define a decimal fixed point type with a delta of 0.1, you will be able to accurately store the values 0.1, 1.0, 2.2, 5.7, etc. You will not be able to accurately store the value 0.01. Instead, the value will be rounded down to 0.0.

If the compiler accepts your fixed point type definition, it guarantees that values represented by that type will have at least the degree of accuracy specified (or better). If the compiler cannot support the type definition (e.g. due to limited hardware) then a compile-time error will result.

Ordinary Fixed Point

For an ordinary fixed point, you just define the delta and a range:

```
delta Delta range Low .. High
```

The delta can be any real value — for example you may define a circle with one arcsecond resolution with:

```
delta 1 / (60 * 60) range 0.0 .. 360.0
```

[There is one rather strange rule about fixed point types: Because of the way they are internally represented, the range might only go up to `'Last - Delta`. This is a bit like a circle — there 0° and 360° is also the same.]

It should be noted that in the example above the smallest possible value used is not $\frac{1}{60^2} = \frac{1}{3600}$. The compiler will choose a smaller value which, by default, is a power of 2 not greater than the delta. In

our example this could be $2^{-12} = \frac{1}{4096}$. In most cases this should render better performance but sacrifices precision for it.

If this is not what you wish and precision is indeed more important, you can choose your own small value via the attribute clause 'Small.

Decimal Fixed Point

You define a decimal fixed point by defining the delta and the number of digits needed:

```
delta Delta digits Num_Digits
```

Delta must be a positive or negative power of 10 — otherwise the declaration is illegal.

```
delta 10 ** (+2) digits 12
delta 10 ** (-2) digits 12
```

If you like, you can also define the range needed:

```
delta Delta_Value digits Num_Digits range Low .. High
```

Differences between Ordinary and Decimal Fixed Point Types

There is an alternative way of declaring a "decimal" fixed point: You declare an ordinary fixed point and use a power of 10 as 'Small. The following two declarations are equivalent with respect to the internal representation:

```
-- decimal fixed point
type Duration is delta 10 ** (-9) digits 9;

-- ordinary fixed point
type Duration is delta 10 ** (-9) range -1.0 .. 1.0;
for Duration'Small use 10 ** (-9);
```

You might wonder what the difference then is between these two declarations. The answer is:

None with respect to precision, addition, subtraction, multiplication with integer values.

The following is an incomplete list of differences between ordinary and decimal fixed point types.

- Decimal fixed point types are intended to reflect typical **COBOL** declarations with a given number of digits. The notion of scaling (taken from COBOL), attribute S'Scale (RM 3.5.10 (11) (Annotated)), is different between decimal and ordinary fixed point.

- Truncation is required for decimal, not for ordinary, fixed point in multiplication and division (RM 4.5.5 (21) (Annotated)) and type conversions. Operations on decimal fixed point are fully specified, which is not true for ordinary fixed point.

- T'Round can be used to specify rounding on conversion (RM 3.5.10 (12) (Annotated)).

- Package Decimal (RM F.2 (Annotated)), which of course applies only to decimal fixed point, defines the decimal Divide generic procedure. If annex F is supported (GNAT does), at least 18 digits must be supported (there is no such rule for fixed point).

- Decimal_IO (RM A.10.1 (73) (Annotated)) has semantics different from Fixed_IO (RM A.10.1 (68) (Annotated)).

- Static expressions must be a multiple of the Small for decimal fixed point.

Conclusion: For normal numeric use, an ordinary fixed point (probably with 'Small defined) should be defined. Only if you are interested in COBOL like use, i.e. well defined deterministic decimal semantics (especially for financial computations, but that might apply to cases other than money) should you take decimal fixed point.

See also

Wikibook

- Ada Programming
- Ada Programming/Types
- Ada Programming/Types/range
- Ada Programming/Types/digits
- Ada Programming/Types/mod
- Ada Programming/Keywords/delta
- Ada Programming/Attributes/'Small

Ada 95 Reference Manual

- 3.5.9 Fixed Point Types (Annotated)

Ada 2005 Reference Manual

- 3.5.9 Fixed Point Types (Annotated)

11 Arrays

An array is a collection of elements which can be accessed by one or more index values. In Ada any definite type is allowed as element and any discrete type, i.e. Range, Modular or Enumeration, can be used as an index.

Declaring arrays

Ada's arrays are quite powerful and so there are quite a few syntax variations, which are presented below.

Basic syntax

The basic form of an Ada array is:

```
array (Index_Range) of Element_Type
```

where Index_Range is a range of values within a discrete index type, and Element_Type is a definite subtype. The array consists of one element of "Element_Type" for each possible value in the given range. If you for example want to count how often aspecific letter appears inside a text, you could use:

```
type Character_Counter is array (Character) of Natural;
```

As a general advice, do not use Integer as the index range, since most of the time negative indices do not make sense. It is also a good style when using numeric indices, to define them starting in 1 instead of 0, since it is more intuitive for humans and avoids off-by-one errors.

With known subrange

Often you don't need an array of all possible values of the index type. In this case you can **subtype** your index type to the actually needed range.

```
subtype Index_Sub_Type is Index_Type range First .. Last
array (Index_Sub_Type) of Element_Type
```

Since this may involve a lot of typing and you may also run out of useful names for new subtypes, the array declaration allows for a shortcut:

```
array (Index_Type range First .. Last) of Element_Type
```

Since `First` and `Last` are expressions of `Index_Type`, a simpler form of the above is:

```
array (First .. Last) of Element_Type
```

Note that if `First` and `Last` are numeric literals, this implies the index type `Integer`.

If in the example above the character counter should only count upper case characters and discard all other characters, you can use the following array type:

```
type Character_Counter is array (Character range 'A' .. 'Z') of Natural;
```

With unknown subrange

Sometimes the range actually needed is not known until runtime or you need objects of different lengths. In lower level languages like C, you would now have to resort to heap memory. Not with Ada. Here we have the box '<>', which allows us to declare indefinite arrays:

```
array (Index_Type range <>) of Element_Type
```

When you declare objects of such a type, the bounds must of course be given and the object is constrained to them.

The predefined type String is such a type. It is defined as

```
type String is array (Positive range <>) of Character;
```

You define objects of such an unconstrained type in several ways (the extrapolation to other arrays than String should be obvious):

```
Text : String (10 .. 20);
Input: String := Read_from_some_file;
```

(These declarations additionally define anonymous subtypes of String.) In the first example, the range of indices is explicitly given. In the second example, the range is implicitly defined from the initial expression, which here could be via a function reading data from some file. Both objects are constrained to their ranges, i.e. they cannot grow nor shrink.

With aliased elements

If you come from C/C++, you are probably used to the fact that every element of an array has an address. The C/C++ standards actually demands that.

In Ada, this is not true. Consider the following array:

```
type Day_Of_Month is range 1 .. 31;
type Day_Has_Appointment is array (Day_Of_Month) of Boolean;
pragma Pack (Day_Has_Appointment);
```

Since we have packed the array, the compiler will use as little storage as possible. And in most cases this will mean that 8 boolean values will fit into one byte.

So Ada knows about arrays where more then one element shares one address. So what if you need to address each single element. Just not using **pragma** *Pack* is not enough. If the CPU has very fast bit access, the compiler might pack the array without being told. You need to tell the compiler that you need to address each element via an access.

```
type Day_Of_Month is range 1 .. 31;
type Day_Has_Appointment is array (Day_Of_Month) of aliased Boolean;
```

Arrays with more than one dimension

Arrays can have more than one index. Consider the following 2-dimensional array:

```
type Character_Display is
   array (Positive range <>, Positive range <>) of Character;
```

This type permits declaring rectangular arrays of characters. Example:

```
Magic_Square: constant Character_Display :=
   (('S', 'A', 'T', 'O', 'R'),
    ('A', 'R', 'E', 'P', 'O'),
    ('T', 'E', 'N', 'E', 'T'),
    ('O', 'P', 'E', 'R', 'A'),
    ('R', 'O', 'T', 'A', 'S'));
```

Or, stating some index values explicitly,

```
Magic_Square: constant Character_Display(1 .. 5, 1 .. 5) :=
   (1 => ('S', 'A', 'T', 'O', 'R'),
    2 => ('A', 'R', 'E', 'P', 'O'),
    3 => ('T', 'E', 'N', 'E', 'T'),
    4 => ('O', 'P', 'E', 'R', 'A'),
    5 => ('R', 'O', 'T', 'A', 'S'));
```

The index values of the second dimension, those indexing the characters in each row, are in 1 .. 5 here. By choosing a different second range, we could change these to be in 11 .. 15:

```
Magic_Square: constant Character_Display(1 .. 5, 11 .. 15) :=
   (1 => ('S', 'A', 'T', 'O', 'R'),
     ...
```

By adding more dimensions to an array type, we could have squares, cubes (or « bricks »), etc., of homogenous data items.

Finally, an array of characters is a string (see Ada Programming/Strings). Therefore, `Magic_Square` may simply be declared like this:

```
Magic_Square: constant Character_Display :=
   ("SATOR",
    "AREPO",
    "TENET",
    "OPERA",
    "ROTAS");
```

Using arrays

Assignment

When accessing elements, the index is specified in parentheses. It is also possible to access slices in this way:

```
Vector_A (1 .. 3) := Vector_B (3 .. 5);
```

Note that the index range slides in this example: After the assignment, Vector_A (1) = Vector_B (3) and similarly for the other indices.

Also note that the ranges overlap, nevertheless Vector_A (3) /= Vector_B (3); a compiler delivering such a result would severely be broken.

Concatenate

The operator "&" can be used to concatenate arrays:

```
Name := First_Name & ' ' & Last_Name;
```

In both cases, if the resulting array does not fit in the destination array, Constraint_Error is raised.

If you try to access an existing element by indexing outside the array bounds, Constraint_Error is raised (unless checks are suppressed).

Array Attributes

There are four Attributes which are important for arrays: 'First, 'Last, 'Length and 'Range. Lets look at them with an example. Say we have the following three strings:

```
Hello_World  : constant String := "Hello World!";
World        : constant String := Hello_World (7 .. 11);
Empty_String : constant String := "";
```

Then the four attributes will have the following values:

Array	'First	'Last	'Length	'Range
Hello_World	1	12	12	1 .. 12
World	7	11	5	7 .. 11
Empty_String	1	0	0	1 .. 0

The example was choosen to show a few common beginner's mistakes

1. The assumption that strings begin with the index value 1 is wrong.
2. The assumption (which follows from the first one) that X'Length = X'Last is wrong.
3. And last the assumption that X'Last > X'First; this is not true for empty strings.

The attribute 'Range is a little special as it does not return a discrete value but an abstract description of the array. One might wonder what it is good for. The most common use is in the [[../../Control#for_loop_on_arrays/|for loop on arrays]] but 'Range can also be used in declaring a name for the index subtype:

```
subtype Hello_World_Index is Integer range Hello_World'Range;
```

The Range attribute can be convenient when programming index checks:

```
if K in World'Range then
   return World(K);
else
   return Substitute;
end if;
```

Empty or Null Arrays

As you have seen in the section above, Ada allows for empty arrays. And — of course — you can have empty arrays of all sorts, not just String:

```
type Some_Array is array (Positive range <>) of Boolean;
Empty_Some_Array : constant Some_Array (1 .. 0) := (others => False);
```

Note: If you give an initial expression to an empty array (which is a must for a constant), the expression in the aggregate will of course not be evaluated since thereare no elements actually stored.

See also

Wikibook

- Ada Programming
- Ada Programming/Types
- Data Structures
- Data Structures/Arrays

Ada 95 Reference Manual

- 3.6 Array Types (Annotated)

Ada 2005 Reference Manual

- 3.6 Array Types (Annotated)

Ada Quality and Style Guide

- 10.5.7 Packed Boolean Array Shifts

12 Records

A **record** is a composite type that groups one or more fields. A field can be of any type, even a record.

Basic record

```
type Basic_Record is
   record
      A : Integer;
   end record;
```

Null record

The null record is when a type without data is needed. There are two ways to declare a null record:

```
type Null_Record is
   record
      null;
   end record;

type Null_Record is null record;
```

For the compiler they are the same. However, programmers often use the first variant if the type is not finished yet to show that they are planning to expand the type later, or they usually use the second if the (tagged) record is a base class in object oriented programming.

Record Values

Values of a record type can be specified using a record aggregate, giving a list of named components thus

```
A_Basic_Record       : Basic_Record          := Basic_Record'(A => 42);
Another_Basic_Record : Basic_Record          := (A => 42);
Nix                  : constant Null_Record := (null record);
```

Given a somewhat larger record type,

```
type Car is record
   Identity       : Long_Long_Integer;
   Number_Wheels  : Positive range 1 .. 10;
   Paint          : Color;
   Horse_Power_kW : Float range 0.0 .. 2_000.0;
   Consumption    : Float range 0.0 .. 100.0;
end record;
```

a value *may* be specified using *positional* notation, that is, specifying a value for each record component in declaration order

```
BMW : Car := (2007_752_83992434, 5, Blue, 190.0, 10.1);
```

However, naming the components of a `Car` aggregate offers a number of advantages.

1. Easy identification of which value is used for which component. (After all, named components are the raison d'être of records.)
2. Reordering the components is allowed—you only have to remember the component names, not their position.
3. Improved compiler diagnostic messages.

Reordering components is possible because component names will inform the compiler (and the human reader!) of the intended value associations. Improved compiler messages are also in consequence of this additional information passed to the compiler. While an omitted component will always be reported due to Ada's coverage rules, messages can be much more specific when there are named associations. Considering the `Car` type from above, suppose a programmer by mistake specifies only one of the two floating point values for `BMW` in positional notation. The compiler, in search of another component value, will then not be able to decide whether the specified value is intended for `Horse_Power_kW` or for `Consumption`. If the programmer instead uses named association, say Horse_Power_kW=> 190.0, it will be clear which other component is missing.

```
BMW : Car :=
  (Identity       => 2007_752_83992434,
   Number_Wheels  => 5,
   Horse_Power_kW => 190.0,
   Consumption    => 10.1,
   Paint          => Blue);
```

Discriminated record

```
type Discriminated_Record (Size : Natural) is
   record
      A : String (1 .. Size);
   end record;
```

Variant record

The variant record is a special type of discriminated record where the presence of some components depend on the value of the discriminant.

```
type Traffic_Light is (Red, Yellow, Green);
type Variant_Record (Option : Traffic_Light) is
   record
      -- common components

      case Option is
         when Red =>
            -- components for red
         when Yellow =>
            -- components for yellow
         when Green =>
            -- components for green
      end case;
   end record;
```

Mutable and immutable variant records

You can declare variant record types such that its discriminant, and thus its variant structure, can be changed during the lifetime of the variable. Such a record is said to be *mutable*. When "mutating" a record, you must assign **all** components of the variant structure which you are mutating at once, replacing the record with a complete variant structure. Although a variant record declaration may allow objects of

its type to be mutable, there are certain restrictions on whether the objects will be mutable. Reasons restricting an object from being mutable include:

- the object is declared with a discriminant (see Immutable_Traffic_Light below)
- the object is aliased (either by use of **aliased** in the object declaration, or by allocation on the heap using **new**)

```
 type Traffic_Light is (Red, Yellow, Green);
 type Mutable_Variant_Record (Option : Traffic_Light := Red) is      -- the discriminant must
have a default value
    record
       -- common components
       Location : Natural;
       case Option is
          when Red =>
             -- components for red
             Flashing : Boolean := True;
          when Yellow =>
             -- components for yellow
             Delay    : Duration;
          when Green =>
             -- components for green
       end case;
    end record;
...
Mutable_Traffic_Light   : Mutable_Variant_Record;                   -- not declaring a
discriminant makes this record mutable
                                                                    -- it has the default
discriminant/variant
                                                                    -- structure and values

Immutable_Traffic_Light : Mutable_Variant_Record (Option => Yellow); -- this record is immutable,
the discriminant cannot be changed
                                                                    -- even though the type
declaration allows for mutable objects
                                                                    -- with different
discriminant values
...
Mutable_Traffic_Light   := (Option => Yellow,                       -- mutation requires
assignment of all components
                            Location => 54,                         -- for the given variant
structure
                            Delay => 2.3);
...
-- restrictions on objects, causing them to be immutable
type Traffic_Light_Access is access Mutable_Variant_Record;
Any_Traffic_Light       : Traffic_Light_Access :=
                             new Mutable_Variant_Record;
Aliased_Traffic_Light   : aliased Mutable_Variant_Record;
```

Conversely, you can declare record types so that the discriminant along with the structure of the variant record may not be changed. To make a record type declaration *immutable*, the discriminant must **not** have a default value.

```
 type Traffic_Light is (Red, Yellow, Green);
 type Immutable_Variant_Record (Option : Traffic_Light) is -- no default value makes the record
type immutable
    record
       -- common components
       Location : Natural := 0;
       case Option is
          when Red =>
             -- components for red
             Flashing : Boolean := True;
          when Yellow =>
             -- components for yellow
             Delay    : Duration;
          when Green =>
             -- components for green
       end case;
    end record;
```

```
...
Default_Traffic_Light   : Immutable_Variant_Record;                    -- ILLEGAL!
Immutable_Traffic_Light : Immutable_Variant_Record (Option => Yellow); -- this record is
immutable, since the type declaration is immutable
```

Union

This language feature is only available in Ada 2005 standard.

```
type Traffic_Light is (Red, Yellow, Green);
type Union (Option : Traffic_Light := Traffic_Light'First) is
   record
      -- common components

      case Option is
         when Red =>
            -- components for red
         when Yellow =>
            -- components for yellow
         when Green =>
            -- components for green
      end case;
   end record;
pragma Unchecked_Union (Union);
pragma Convention (C, Union);    -- optional
```

The difference to a variant record is such that *Option* is not actually stored inside the record and never checked for correctness - it's just a dummy.

This kind of record is usually used for interfacing with C but can be used for other purposes as well (then without `pragma Convention (C, Union);`).

Tagged record

The tagged record is one part of what in other languages is called a class. It is the basic foundation of object orientated programming in Ada. The other two parts a class in Ada needs is a package and primitive operations.

```
type Person is tagged
   record
      Name   : String (1 .. 10);
      Gender : Gender_Type;
   end record;


type Programmer is new Person with
   record
      Skilled_In : Language_List;
   end record;
```

Ada 2005 only:

```
type Programmer is new Person
                   and Printable
with
   record
      Skilled_In : Language_List;
   end record;
```

Abstract tagged record

An abstract type has at least an abstract primitive operation, i.e. one of its operations is not defined and then its derivative types has to provide an implementation.

With aliased elements

If you come from C/C++, you are probably used to the fact that every element of a record - which is not part of a bitset - has an address. In Ada, this is not true because records, just like arrays, can be packed. And just like arrays you can use **aliased** to ensure that an element can be accessed via an access type.

```
type Basic_Record is
   record
      A : aliased Integer;
   end record ;
```

Please note: each element needs its own **aliased**.

Limited Records

In addition to being variant, tagged, and abstract, records may also be limited (no assignment, and no predefined equality operation for Limited Types). In object oriented programming, when tagged objects are handled by references instead of copying them, this blends well with making objects limited.

See also

Wikibook

- Ada Programming
- Ada Programming/Types
- Ada Programming/Keywords/record
- Ada Programming/Keywords/null
- Ada Programming/Keywords/abstract
- Ada Programming/Keywords/case
- Ada Programming/Keywords/when
- Ada Programming/Pragmas/Unchecked_Union

Ada Reference Manual

Ada 95

- 3.8 Record Types (Annotated)

Ada 2005

- 3.8 Record Types (Annotated)
- Annex B.3.3 Pragma Unchecked_Union (Annotated)

Ada Issues

- AI-00216 Unchecked unions -- variant records with no run-time discriminant

13 Access types

Different access types

Pool access

A pool access type handles accesses to objects which were created on some specific heap (or storage pool as it is called in Ada). It may not point to a stack or library level (static) object or an object in a different storage pool.

```
type Day_Of_Month is range 1 .. 31;
type Day_Of_Month_Access is access Day_Of_Month;
```

The storage pool is implementation defined if not specified. Remember that Ada supports user defined storage pools, so you can define the storage pool with

```
for Day_Of_Month_Access'Storage_Pool use Pool_Name;
```

Access all

Handles an access to an object which was created on any storage pool, on the stack or at library level (static).

```
type Day_Of_Month is range 1 .. 31;
type Day_Of_Month_Access is access all Day_Of_Month;
```

Access constant

Handles an access either to a constant or variable object which was created on any storage pool, on the stack or at library level (static), but with a restriction to read-only usage of the referenced object, whether it be intrinsically a constant or a variable.

```
type Day_Of_Month is range 1 .. 31;
type Day_Of_Month_Access is access constant Day_Of_Month;
```

Anonymous access

An anonymous access is used as a parameter to a function, procedure or a discriminated type. Here some examples:

```
procedure Test (Some_Day : access Day_Of_Month);

task type Thread (
   Execute_For_Day : access Day_Of_Month)
is
   --  ...In Ada 2005 a
end Thread;

type Day_Data (
   Store_For_Day : access Day_Of_Month)
```

```
is record
   --  components
end record;
```

Before you use an anonymous access you should consider if the "**out**" or "**in out**" modifier is not more appropriate - see access performance to see why.

This language feature is only available in Ada 2005 standard.

In Ada 2005 an anonymous access is also allowed in more circumstances:

```
type Object is record
   M   : Integer;
   Next: access Object;
end record;
X: access Integer;
function F return access constant Float;
```

Not null access

This language feature is only available in Ada 2005 standard.

All access types can be modified with "**not null**":

```
type Day_Of_Month_Access is access Day_Of_Month;
subtype Day_Of_Month_Not_Null_Access is not null Day_Of_Month_Access;
```

The type must then always point to an object, so initializations are compulsory.

The compiler also allows you to declare a *type* directly as "not null access" instead of as a *subtype* of other access type:

```
type Day_Of_Month_Access is not null access Day_Of_Month;
```

However, in nearly all cases this is not a good idea because it would be nearly unusable (for example, you would be unable to free the allocated memory). Not null access are intended for access *subtypes*, object *declarations*, and subprogram *parameters*.[2]

Constant anonymous access

This language feature is only available in Ada 2005 standard.

An anonymous access to a constant object.

```
procedure Test (Some_Day : access constant Day_Of_Month);
```

Access to subprogram

An access to subprogram allows us to call a subprogram without knowing its name nor its declaration location. One of the uses of this kind of access is the well known callbacks.

```
type Callback_Procedure is access procedure (Id   : Integer;
                                             Text: String);
```

```
type Callback_Function is access function (The_Alarm: Alarm) return Natural;
```

For getting an access to a subprogram, the attribute Access is applied to a subprogram name with the proper parameter and result profile.

```
procedure Process_Event (Id   : Integer;
                         Text: String);
My_Callback : Callback_Procedure := Process_Event'Access;
```

Anonymous access to subprogram

This language feature is only available in Ada 2005 standard.

```
procedure Test (
   Call_Back : access procedure (
                         Id   : Integer;
                         Text : String));
```

There is now no limit on the number of keyword in a sequence:

```
function F return access function return access function return access Some_Type;
```

This is a function that returns the access to a function that in turn returns an access to a function returning an access to some type.

Using access types

Creating object in a storage pool

Objects in a storage pool are created with the keyword **new**:

```
Father : Person_Access := new Person'(Father_First_Name, Father_Last_Name);
```

Deleting object from a storage pool

Although the Ada standard mentioned the use of a garbage collector which would automatically remove all unneeded objects that had been created on the heap (storage pool), only Ada compilers targeting a virtual machine like Java or .NET actually have garbage collectors.

Therefore in order to delete an object from the heap you will need the generic unit Ada.Unchecked_Deallocation.

```
with Ada.Unchecked_Deallocation;
procedure Deallocation_Sample is

   type Vector is array (Integer range <>) of Float;
   type Vector_Ref is access Vector;
   procedure Free_Vector is new Ada.Unchecked_Deallocation
      (Object => Vector, Name => Vector_Ref);

   VA, VB: Vector_Ref;
   V     : Vector;

begin
   VA     := new Vector (1 .. 10);
```

```
      VB     := VA;  -- points to the same location as VA
      VA.all := (others => 0.0);
      --  ... Do whatever you need to do with the vector
      Free_Vector (VA); -- The memory is deallocated and VA is now null
      V := VB.all;  -- VB is not null, access to a dangling pointer is erroneous
   end Deallocation_Sample;
```

Because of the problems with dangling pointer, this operation is called **unchecked** deallocation. It is the chore of the programmer to take care that this does not happen.

Since Ada does allow for user defined storage pools you could also try a garbage collector library.

Implicit dereferencing

If you declare a record type, and an access type for it, thus:

```
type Person is record
   First_Name : String (1..30);
   Last_Name  : String (1..20);
end record;
type Person_Access is access Person;
```

you can use these types as follows:

```
Father : Person_Access := new Person'(Father_First_Name, Father_Last_Name);
...
Ada.Text_IO.Put(Father.all.First_Name);
```

However, *.all* is implicit if omitted, so we can write more concisely:

```
Ada.Text_IO.Put(Father.First_Name);
```

and be careful:

```
Obj1 : Person_Access := new Person'(...);
Obj2 : Person_Access := new Person'(...);
Obj1.all := Obj2.all;  -- Obj1 still refers to an object different from Obj2,
                       -- but it has the same content (deep copy).
Obj1 := Obj2;          -- Obj1 now refers to the same person as Obj2 (shallow copy)
```

Implicit dereferencing also applies to arrays:

```
type Vector is array (0..3) of Complex;
type Vector_Access is access Vector;
VA : Vector_Access := new Vector;
...
C : Complex := VA(3); -- a shorter equivalent for VA.all(3)
```

Access FAQ

A few "Frequently Asked Question" and "Frequently Encountered Problems" (mostly from former C users) regarding Ada's access types.

Access vs. access all

An **access all** can do anything a simple **access** can do. So one might ask: "Why use **access** at all?" -

And indeed some programmers never use simple **access**.

But this is wrong because **access all** is more error prone! One example: When Ada.Unchecked_Deallocation is used on an **access all** it is not checked that the object actually belongs to the proper storage pool - or if the object belongs to any storage pool at all. The following example, invalidly and perhaps disastrously, will try to deallocate a stack object:

```
declare
   type Day_Of_Month is range 1 .. 31;
   type Day_Of_Month_Access is access all Day_Of_Month;
  procedure Free
  is new Ada.Unchecked_Deallocation (
      Object => Day_Of_Month
      Name   => Day_Of_Month_Access);
  A : aliased Day_Of_Month;
  Ptr : Day_Of_Month_Access := A'Access;
begin
  Free(Ptr);
end;
```

With a simple **access** you know at least that you won't try to deallocate a stack object.

Depending on the implementation an **access** might also render better performance then **access all**.

Access performance

Ada performs run-time checks on access types, so not all access types render the same performance. In other places of the Ada Programming we suggest that you should not care about runtime checks since the optimizer will remove them when they are not needed. However former C programmers often use anonymous access — which have worse performance — instead of using "out" or "in out" parameter modes which have best performance. But note that parameter modes "in", "in out", "out" have nothing to do with parameter passing mechanisms "by copy", "by reference" etc; they only describe the data flow direction. The compiler is therefore free to choose whatever mechanism is the best for the data type and mode.

Here is a performance list — including the access modifiers — from best to worse:

1. "in" modifier : parameter is passed into the subprogram.
2. "out" modifier : parameter is passed from the subprogram.
3. "in out" modifier : parameter may be modified by the subprogram.
4. pool access : the access is always within a storage pool and only checks for not null are performed upon dereference.
5. access all : additional checks may be needed.
6. anonymous access : complex "stack deeps level" checks are needed so an "anonymous access" can be safely converted from and to an "access all" or "pool access".

Access vs. System.Address

An access is not a memory address. It is something more. For example, an "access to String" often needs some way of storing the string size as well. If you need a simple address and are not concerned about strong-typing, you may consider using the System.Address type.

C compatible pointer

The correct way to create a C compatible access is to use **pragma** Convention:

```
type Day_Of_Month is range 1 .. 31;
for Day_Of_Month'Size use Interfaces.C.int'Size;
pragma Convention (
   Convention => C,
   Entity     => Day_Of_Month);
type Day_Of_Month_Access is access Day_Of_Month;
pragma Convention (
   Convention => C,
   Entity     => Day_Of_Month_Access);
```

pragma Convention should be used on any type you want to use in C. The compiler should warn you if the type cannot be made C compatible.

You may also consider the following - shorter - alternative when declaring Day_Of_Month:

```
type Day_Of_Month is new Interfaces.C.int range 1 .. 31;
```

Before you use access types in C you should consider using the normal "in", "out" and "in out" modifiers. **pragma** Export and **pragma** Import know how parameters are usually passed in C and will use a pointer to pass a parameter automatically where a "normal" C programmer would have used them as well.

Of course the definition of a "normal" C programmer is not left to change, the RM contains precise rules on when to use a pointer for "in", "out", and "in out" - see "B.3 Interfacing with C (Annotated)".

Where is void*?

While actually been a problem for "interfacing with C", here is some possible solutions:

```
procedure Test is

  subtype Pvoid is System.Address;
  -- the declaration in C looks like this:
  -- int C_fun(int *)
  function C_fun (pv: Pvoid) return Integer;
  pragma Import (Convention    => C,
                 Entity        => C_fun,     -- any Ada name
                 External_Name => "C_fun");  -- the C name
  Pointer: Pvoid;
  Input_Parameter: aliased Integer := 32;
  Return_Value   : Integer;
begin

  Pointer      := Input_Parameter'Address;
  Return_Value := C_fun (Pointer);
end Test;
```

Less portable but perhaps more usable (for 32 bit CPUs):

```
type void is mod 2 ** 32;
for void'Size use 32;
```

With GNAT you can get 32/64 bit portability by using:

```
type void is mod System.Memory_Size;
for void'Size use System.Word_Size;
```

Closer to the true nature of void - pointing to an element of zero size is a pointer to a null record. This also has the advantage of having a representation for `void` and `void*`:

```
type Void is null record;
pragma Convention (C, Void);
type Void_Ptr is access all Void;
pragma Convention (C, Void_Ptr);
```

See also

Wikibook

- Ada Programming
- Ada Programming/Types

Ada Reference Manual

Ada 95

- 4.8 Allocators (Annotated)
- 13.11 Storage Management (Annotated)
- 13.11.2 Unchecked Storage Deallocation (Annotated)
- 3.7 Discriminants (Annotated)
- 3.10 Access Types (Annotated)
- 6.1 Subprogram Declarations (Annotated)
- B.3 Interfacing with C (Annotated)

Ada 2005

- 4.8 Allocators (Annotated)
- 13.11 Storage Management (Annotated)
- 13.11.2 Unchecked Storage Deallocation (Annotated)
- 3.7 Discriminants (Annotated)
- 3.10 Access Types (Annotated)
- 6.1 Subprogram Declarations (Annotated)
- B.3 Interfacing with C (Annotated)

Ada Quality and Style Guide

- 5.4.5 Dynamic Data
- 5.9.2 Unchecked Deallocation

14 LIMITED TYPES

Limited Types

When a type is declared **limited** this means that objects of the type cannot be assigned values of the same type. An Object b of limited type LT cannot be copied into an object a of same type LT.

Additionally, there is no predefined equality operation for objects of a limited type.

The desired effects of declaring a type limited include prevention of shallow copying. Also, the (unique) identity of an object is retained: once declared, a name of a variable of type LT will continue to refer to the same object.

The following example will use a rather simplifying type Boat.

```
type Boat is limited private;

function Choose
  (Load  : Sailors_Units;
   Speed : Sailors_Units)
   return  Boat;

procedure Set_Sail (The_Boat : in out Boat);
```

When we declare a variable to be of type Boat, its name will denote one boat from then on. Boats will not be copied into one another.

The full view of a boat might be implemented as a record such as

```
type Boat is limited record
   Max_Sail_Area : Sailors_Units;
   Max_Freight   : Sailors_Units;
   Sail_Area     : Sailors_Units;
   Freight       : Sailors_Units;
end record;
```

The Choose function returns a Boat object depending on the parameters Load and Speed. If we now declare a variable of type Boat we will be better off Choosing an initial Boat (or else we might be dropping into uninitialized waters!). But when we do so, the initialization looks suspiciously like assignment which is not available with limited types:

```
procedure Travel (People : Positive; Average_Speed : Sailors_Units) is

   Henrietta : Boat :=   -- assignment?
      Choose
        (Load  => People * Average_Weight * 1.5,
         Speed => Average_Speed * 1.5);

begin
   Set_Sail (Henrietta);
end Travel;
```

Fortunately, current Ada distinguishes initialization from copying. Objects of a limited type may be initialized by an initialization expression on the right of the delimiter :=.

(Just to prevent confusion: The Ada Reference Manual discriminates between *assignment* and *assignment statement*, where assignment is part of the assignment statement. An initialisation is of course an assignment which, for limited types, is done *in place*. An assignment statement involves copying,

which is forbidden for limited types.)

Related to this feature are aggregates of limited types and "constructor functions" for limited types. Internally, the implementation of the `Choose` function will return a limited record. However, since the return type `Boat` is limited, there must be no copying anywhere. Will this work? A first attempt might be to declare a `result` variable local to `Choose`, manipulate `result`, and return it. The `result` object needs to be "transported" into the calling environment. But `result` is a variable local to `Choose`. When `Choose` returns, `result` will no longer be in scope. Therefore it looks like `result` must be copied but this is not permitted for limited types. There are two solutions provided by the language: extended return statements (see 6.5 Return Statements (Annotated)) and aggregates of limited types. The following body of `Choose` returns an aggregate of limited type `Boat`, after finding the initial values for its components.

```
function Choose
  (Load  : Sailors_Units;
   Speed : Sailors_Units)
   return  Boat
is
   Capacity : constant Sailors_Units := Capacity_Needed (Load);
begin
   return Boat'
     (Max_Freight   => Capacity,
      Max_Sail_Area => Sail_Needed (Capacity),
      Freight       => Load,
      Sail_Area     => 0.0);
end Choose;
```

The object that is returned is at the same time the object that is to have the returned value. The function therefore initializes `Henrietta` *in place*.

In parallel to the predefined type Ada.Finalization.`Controlled`, Ada provides the type `Limited_Controlled` in the same package. It is a limited version of the former.

Initialising Limited Types

A few methods to initialise such types are presented.

```
package Limited_Private_Samples is

  type Uninitialised  is limited private;
  type Preinitialised is limited private;
  type Dynamic_Initialisation is limited private;
  function Constructor (X: Integer)  -- any kind of parameters
    return Dynamic_Initialisation;
  type Needs_Constructor (<>) is limited private;
  function Constructor (X: Integer)  -- any kind of parameters
    return Needs_Constructor;
private

  type Uninitialised is record
    I: Integer;
  end record;
  type Preinitialised is record
    I: Integer := 0;  -- can also be a function call
  end record;
  type Void is null record;
  function Constructor (Object: access Dynamic_Initialisation) return Void;
  type Dynamic_Initialisation is limited record
    Hook: Void := Constructor (Dynamic_Initialisation'Access);
    Bla : Integer;  -- any needed components
  end record;
  type Needs_Constructor is record
    I: Integer;
  end record;
end Limited_Private_Samples;
```

```
package body Limited_Private_Samples is

  function Constructor (Object: access Dynamic_Initialisation) return Void is
  begin
    Object.Bla := 5;  -- may be any value only known at run time
    return (null record);
  end Constructor;
  function Constructor (X: Integer) return Dynamic_Initialisation is
  begin
    return (Hook => (null record),
            Bla  => 42);
  end Constructor;
  function Constructor (X: Integer) return Needs_Constructor is
  begin
    return (I => 42);
  end Constructor;
end Limited_Private_Samples;


 with Limited_Private_Samples;
 use  Limited_Private_Samples;

 procedure Try is

   U: Uninitialised;   -- very bad
   P: Preinitialised;  -- has initial value (good)

   D1: Dynamic_Initialisation;  -- has initial value (good)
   D2: Dynamic_Initialisation := Constructor (0);  -- Ada 2005 initialisation
   D3: Dynamic_Initialisation renames Constructor (0);  -- already Ada 95

   -- I: Needs_Constructor;  -- Illegal without initialisation
   N: Needs_Constructor := Constructor (0);  -- Ada 2005 initialisation

 begin

  null;

 end Try;
```

Note that D3 is a constant, whereas all others are variables.

Also note that the initial value that is defined for the component of Preinitialised is evaluated at the time of object creation, i.e. if an expression is used instead of the literal, the value can be run-time dependent.

```
X, Y: Preinitialised;
```

In the above example, the initial expression would be evaluated twice and could deliver different values. However the sequence of initialisation is undefined, i.e. it's not defined which of X and Y is initialised first.

```
X: Preinitialised;
Y: Preinitialised;
```

Here, X is initialised before Y.

See also

Ada 95 Reference Manual

- 7.5 Limited Types (Annotated)

Ada 2005 Reference Manual

- 7.5 Limited Types (Annotated)

Ada Quality and Style Guide

- 5.3.3 Private Types
- 8.3.3 Formal Private and Limited PrivateTypes

15 Strings

Ada supports three different types of strings. Each string type is designed to solve a different problem.

In addition, every string type is implemented for each available Characters type (Character, Wide_Character, Wide_Wide_Character) giving a complement of nine combinations.

Fixed-length string handling

Fixed-Length Strings are arrays of Character, and consequently of a fixed length. Since a fixed length string is an indefinite subtype the length does not need to be known at compile time - the length may well be calculated at run time. In the following example the length is calculated from command-line argument 1:

```
X : String := Ada.Command_Line.Argument (1);
```

However once the length has been calculated and the strings have been created the length stays constant. Try the following program which shows a typical mistake:

File: show_commandline_1.adb (view, plain text, download page, browse all)

```
with Ada.Text_IO;
with Ada.Command_Line;
procedure Show_Commandline_1 is

   package T_IO renames Ada.Text_IO;
   package CL   renames Ada.Command_Line;
   X : String := CL.Argument (1);
begin
   T_IO.Put ("Argument 1 = ");
   T_IO.Put_Line (X);

   X := CL.Argument (2);

   T_IO.Put ("Argument 2 = ");
   T_IO.Put_Line (X);
end Show_Commandline_1;
```

The program will only work when the 1st and 2nd parameter have the same length. This is even true when the 2nd parameter is shorter. There is neither an automatic padding of shorter strings nor an automatic truncation of longer strings.

Having said that, the package Ada.Strings.Fixed contains a set of procedures and functions for Fixed-Length String Handing which allows padding of shorter strings and truncation of longer strings.

Try the following example to see how it works:

File: show_commandline_2.adb (view, plain text, download page, browse all)

```
with Ada.Text_IO;
with Ada.Command_Line;
with Ada.Strings.Fixed;
procedure Show_Commandline_2 is

   package T_IO renames Ada.Text_IO;
   package CL   renames Ada.Command_Line;
   package S    renames Ada.Strings;
   package SF   renames Ada.Strings.Fixed;
```

```
   X : String := CL.Argument (1);
begin
   T_IO.Put ("Argument 1 = ");
   T_IO.Put_Line (X);

  SF.Move (
     Source  => CL.Argument (2),
     Target  => X,
     Drop    => S.Right,
     Justify => S.Left,
     Pad     => S.Space);

   T_IO.Put ("Argument 2 = ");
   T_IO.Put_Line (X);
end Show_Commandline_2;
```

Bounded-length string handling

Bounded-Length Strings can be used when the maximum length of a string is known and/or restricted. This is often the case in database applications where only a limited amount of characters can be stored.

Like Fixed-Length Strings the maximum length does not need to be known at compile time - it can also be calculated at runtime - as the example below shows:

File: show_commandline_3.adb (view, plain text, download page, browse all)

```
with Ada.Text_IO;
with Ada.Command_Line;
with Ada.Strings.Bounded;
procedure Show_Commandline_3 is

   package T_IO renames Ada.Text_IO;
   package CL   renames Ada.Command_Line;
   function Max_Length (
      Value_1 : Integer;
      Value_2 : Integer)
   return
      Integer
   is
      Retval : Integer;
   begin
      if Value_1 > Value_2 then
         Retval := Value_1;
      else
         Retval := Value_2;
      end if;
      return Retval;
   end Max_Length;
   pragma Inline (Max_Length);
   package SB
   is new Ada.Strings.Bounded.Generic_Bounded_Length (
       Max => Max_Length (
                  Value_1 => CL.Argument (1)'Length,
                  Value_2 => CL.Argument (2)'Length));
   X :  SB.Bounded_String
     := SB.To_Bounded_String (CL.Argument (1));
begin
   T_IO.Put ("Argument 1 = ");
   T_IO.Put_Line (SB.To_String (X));

   X := SB.To_Bounded_String (CL.Argument (2));

   T_IO.Put ("Argument 2 = ");
   T_IO.Put_Line (SB.To_String (X));
end Show_Commandline_3;
```

You should know that Bounded-Length Strings have some distinct disadvantages. Most noticeable is that each Bounded-Length String is a different type which makes converting them rather cumbersome. Also a Bounded-Length String type always allocates memory for the maximum permitted string length

for the type. The memory allocation for a Bounded-Length String is equal to the maximum number of string "characters" plus an implementation dependent number containing the string length (each character can require allocation of more than one byte per character, depending on the underlying character type of the string, and the length number is 4 bytes long for the Windows GNAT Ada compiler v3.15p, for example).

Unbounded-length string handling

Last but not least there is the Unbounded-Length String. In fact: If you are not doing embedded or database programming this will be the string type you are going to use most often as it gives you the maximum amount of flexibility.

As the name suggest the Unbounded-Length String can hold strings of almost any length - limited only to the value of Integer'Last or your available heap memory.

File: show_commandline_4.adb (view, plain text, download page, browse all)

```
with Ada.Text_IO;
with Ada.Command_Line;
with Ada.Strings.Unbounded;
procedure Show_Commandline_4 is

   package T_IO renames Ada.Text_IO;
   package CL   renames Ada.Command_Line;
   package SU   renames Ada.Strings.Unbounded;
   X :  SU.Unbounded_String
     := SU.To_Unbounded_String (CL.Argument (1));
begin
   T_IO.Put ("Argument 1 = ");
   T_IO.Put_Line (SU.To_String (X));

   X := SU.To_Unbounded_String (CL.Argument (2));

   T_IO.Put ("Argument 2 = ");
   T_IO.Put_Line (SU.To_String (X));
end Show_Commandline_4;
```

As you can see the Unbounded-Length String example is also the shortest (discarding the first example - which is buggy) - this makes using Unbounded-Length Strings very appealing.

See also

Wikibook

- Ada Programming

Ada 95 Reference Manual

- 2.6 String Literals (Annotated)
- 3.6.3 String Types (Annotated)
- A.4.3 Fixed-Length String Handling (Annotated)
- A.4.4 Bounded-Length String Handling (Annotated)
- A.4.5 Unbounded-Length String Handling (Annotated)

Ada 2005 Reference Manual

- 2.6 String Literals (Annotated)
- 3.6.3 String Types (Annotated)
- A.4.3 Fixed-Length String Handling (Annotated)
- A.4.4 Bounded-Length String Handling (Annotated)
- A.4.5 Unbounded-Length String Handling (Annotated)

16 SUBPROGRAMS

In Ada the subprograms are classified into two categories: procedures and functions. A procedures call is a statement and does not return any value, whereas a function returns a value and must therefore be a part of an expression.

Subprogram parameters may have four modes.

in
: The actual parameter value goes into the call and is not changed there.The formal parameter is a constant and allows only reading. This is the default when no modeis given. The actual parameter is an expression.

in out
: The actual parameter goes into the call and may be redefined. The formal parameter is a variable and can be read and written.

out
: The actual parameter's value before the call is irrelevant,it will get a value in the call. The formal parameter can be read and written. (In Ada 83 **out** parameters were write-only.)

access
: The formal parameter is an access (a pointer) to some variable.

Note that parameter modes do not specify the parameter passing method. Their purpose is to document the data flow.

The parameter passing method depends on the type of the parameter. A rule of thumb is that parameters fitting into a register are passed by copy, others are passed by reference. For certain types, there are special rules, for others the parameter passing mode is left to the compiler.

Unlike other programming languages, Ada does not need an empty set of ( ) when a subprogram has no parameters.

Procedures

A procedure call in Ada constitutes a statement by itself.

For example:

```
procedure A_Test (A, B: in Integer; C: out Integer) is
begin
   C := A + B;
end A_Test;
```

When the procedure is called with the statement

```
A_Test (5 + P, 48, Q);
```

the expressions 5 + P and 48 are evaluated (expressions are only allowed for in parameters), and then assigned to the formal parameters A and B, which behave like constants. Then, the value A + B is assigned to formal variable C, whose value will be assigned to the actual parameter Q when the procedure finishes.

C, being an **out** parameter, is an unitialized variable before the first assignment. (Therefore in Ada 83, there existed the restriction that **out** parameters are write-only. If you wanted to read the value written, you had to declare a local variable, do all calculations with it, and finally assign it to C before return. This was awkward and error prone so the restriction was removed in Ada 95.)

Within a procedure, the return statement can be used without arguments to exit the procedure and return the control to the caller.

For example, to solve an equation of the kind $ax^2 + bx + c = 0$:

```
with Ada.Numerics.Elementary_Functions;
use  Ada.Numerics.Elementary_Functions;
procedure Quadratic_Equation
   (A, B, C :     Float;    -- By default it is "in".
    R1, R2  : out Float;
    Valid   : out Boolean)
is
   Z : Float;
begin
   Z := B**2 - 4.0 * A * C;
   if Z < 0.0 or A = 0.0 then
      Valid := False;  -- Being out parameter, it must be modified at least once.
      R1    := 0.0;
      R2    := 0.0;
   else
      Valid := True;
      R1    := (-B + Sqrt (Z)) / (2.0 * A);
      R2    := (-B - Sqrt (Z)) / (2.0 * A);
   end if;
end Quadratic_Equation;
```

The function SQRT calculates the square root of non-negative values. If the roots are real, they are given back in R1 and R2, but if they are complex or the equation degenerates (A = 0), the execution of the procedure finishes after assigning to the Valid variable the False value, so that it is controlled after the call to the procedure. Notice that the **out** parameters should be modified at least once, and that if a mode is not specified, it is implied **in**.

Functions

A function is a subprogram that can be invoked as part of an expression. Functions can only take **in** (the default) or **access** parameters. The latter can be used as a work-around for the restriction that functions may not have **out** parameters. In this sense, Ada functions behave more like mathematical functions than in other languages. The specification of the function is necessary to show to the clients all the information needed to invoke it.

A function body example can be:

```
function Minimum (A, B : Integer) return Integer is
begin
   if A <= B then
      return A;
   else
      return B;
   end if;
end Minimum;
```

The formal parameters of a function behave as local constants whose values are provided by the corresponding actual parameters. The statement **return** is used to indicate the value returned by the function call and to give back the control to the expression that called the function. The expression of the

return statement may be of arbitrary complexity and must be of the same type declared in the specification. If an incompatible type is used, the compiler gives an error. If the restrictions of a subtype are not fulfilled, e.g. a range, it raises a Constraint_Error exception.

The body of the function can contain several **return** statements and the execution of any of them will finish the function, returning control to the caller. If the flow of control within the function branches in several ways, it is necessary to make sure that each one of them is finished with a **return** statement. If at run time the end of a function is reached without encountering a **return** statement, the exception Program_Error is raised. Therefore, the body of a function must have at least one such **return** statement.

Every call to a function produces a new copy of any object declared within it. When the function finalizes, its objects disappear. Therefore, it is possible to call the function recursively. For example, consider this implementation of the factorial function:

```
function Factorial (N : Positive) return Positive is
begin
   if N = 1 then
      return 1;
   else
      return (N * Factorial (N - 1));
   end if;
end Factorial;
```

When evaluating the expression `Factorial (4);` the function will be called with parameter 4 and within the function it will try to evaluate the expression `Factorial (3)`, calling itself as a function, but in this case parameter N would be 3 (each call copies the parameters) and so on until N = 1 is evaluated which will finalize the recursion and then the expression will begin to be completed in the reverse order.

A formal parameter of a function can be of any type, including vectors or records. Nevertheless, it cannot be an anonymous type, that is, its type must be declared before, for example:

```
type Float_Vector is array (Positive range <>) of Float;
function Add_Components (V: Float_Vector) return Float is
   Result : Float := 0.0;
begin
   for I in V'Range loop
      Result := Result + V(I);
   end loop;
   return Result;
end Add_Components;
```

In this example, the function can be used on a vector of arbitrary dimension. Therefore, there are no static bounds in the parameters passed to the functions. For example, it is possible to be used in the following way:

```
V4  : Float_Vector (1 .. 4) := (1.2, 3.4, 5.6, 7.8);
Sum : Float;
Sum := Add_Components (V4);
```

In the same way, a function can also return a type whose bounds are not known a priori. For example:

```
function Invert_Components (V : Float_Vector) return Float_Vector is
   Result : Float_Vector(V'Range);    -- Fix the bounds of the vector to be returned.
begin
   for I in V'Range loop
      Result(I) := V (V'First + V'Last - I);
   end loop;
   return Result;
end Invert_Components;
```

The variable Result has the same bounds as V, so the returned vector will always have the same dimension as the one passed as parameter.

A value returned by a function can be used without assigning it to a variable, it can be referenced as an expression. For example, `Invert_Components (V4) (1)`, where the first element of the vector returned by the function would be obtained (in this case, the last element of V4, i.e. 7.8).

Named parameters

In subprogram calls, named parameter notation (i.e. the name of the formal parameter followed of the symbol => and then the actual parameter) allows the rearrangement of the parameters in the call. For example:

```
Quadratic_Equation (Valid => OK, A => 1.0, B => 2.0, C => 3.0, R1 => P, R2 => Q);
F := Factorial (N => (3 + I));
```

This is expecially useful to make clear which parameter is which.

```
Phi := Arctan (A, B);
Phi := Arctan (Y => A, X => B);
```

The first call (from Ada.Numerics.Elementary_Functions) is not very clear. One might easily confuse the parameters. The second call makes the meaning clear without any ambiguity.

Another use is for calls with numeric literals:

```
Ada.Float_Text_IO.Put_Line (X, 3, 2, 0);  -- ?
Ada.Float_Text_IO.Put_Line (X, Fore => 3, Aft => 2, Exp => 0);  -- OK
```

Default parameters

On the other hand, formal parameters may have default values. They can, therefore, be omitted in the subprogram call. For example:

```
procedure By_Default_Example (A, B: in Integer := 0);
```

can be called in these ways:

```
By_Default_Example (5, 7);      -- A = 5, B = 7
By_Default_Example (5);         -- A = 5, B = 0
By_Default_Example;             -- A = 0, B = 0
By_Default_Example (B => 3);    -- A = 0, B = 3
By_Default_Example (1, B => 2); -- A = 1, B = 2
```

In the first statement, a "regular call" is used (with positional association); the second also uses positional association but omits the second parameter to use the default; in the third statement, all parameters are by default; the fourth statement uses named association to omit the first paramer; finally, the fifth statement uses mixed association, here the positional parameters have to precede the named ones.

Note that the default expression is evaluated once for each formal parameter that has no actual parameter. Thus, if in the above example a function would be used as defaults for A and B, the function

would be evaluated once in case 2 and 4; twice in case 3, so A and B could have different values; in the others cases, it would not be evaluated.

See also

Wikibook

- Ada Programming
- Ada Programming/Operators

Ada 95 Reference Manual

- Section 6: Subprograms (Annotated)
- 4.4 Expressions (Annotated)

Ada 2005 Reference Manual

- Section 6: Subprograms (Annotated)
- 4.4 Expressions (Annotated)

Ada Quality and Style Guide

- 4.1.3 Subprograms

17 Packages

One of the biggest advantages of Ada over most other programming languages is its well defined system of modularization and separate compilation. Even though Ada allows separate compilation, it maintains the strong type checking among the various compilations by enforcing rules of compilation order and compatibility checking. Ada uses separate compilation (like Modula-2, Java and C#), and not independent compilation (as C/C++ does), in which the various parts are compiled with no knowledge of the other compilation units with which they will be combined.

A note to C/C++ users: Yes, you can use the preprocessor to emulate separate compilation -- but it is only an emulation and the smallest mistake leads to very hard to find bugs. It is telling that all C/C++ successor languages including D have turned away from the independent compilation and the use of the preprocessor.

So it's good to know that Ada has had separate compilation ever since Ada-83 and is probably the most sophisticated implementation around.

The following is a quote from the current Ada Reference Manual RM 7(1)

"Packages are program units that allow the specification of groups of logically related entities. Typically, a package contains the declaration of a type (often a private type or private extension) along with the declaration of primitive subprograms of the type, which can be called from outside the package, while their inner workings remain hidden from outside users."

Parts of a package

This section is a stub.
You can help Wikibooks by expanding it.

A package consists of 3 parts but only the specification is mandatory: you leave out the body and the private part if you don't need them - you can have package specification without package body and the private package specification is optional.

The public package specification

The package specification of an Ada package describes all the subprogram specifications, variables, types, constants etc that are visible to anyone who wishes to use the package.

```
package Public_Only_Package is

  type Range_10 is range 1 .. 10;
end Public_Only_Package;
```

The private package specification

The private part of a package serves two purposes:

- To complete the defered definition of private types and constants.
- To export entities only visible to the children of the package

```
package Package_With_Private is

   type Private_Type is private;
private

   type Private_Type is array (1 .. 10) of Integer;
end Package_With_Private;
```

The package body

The package body defines the implementation of the package. All the subprograms defined in the specification have to be implemented in the body. New subprograms, types and objects can be defined in the body that are not visible to the users of the package.

```
package Package_With_Body is

   type Basic_Record is private;
   procedure Set_A (This : in out Basic_Record;
                    An_A : in     Integer);
   function Get_A (This : Basic_Record) return Integer;
private

   type Basic_Record is
      record
         A : Integer;
      end record ;
   pragma Pure_Function  (Get_A);
   pragma Inline (Get_A);
   pragma Inline (Set_A);
end Package_With_Body;


package body Package_With_Body is

   procedure Set_A (This : in out Basic_Record;
                    An_A : in     Integer)
   is
   begin
      This.A := An_A;
   end Set_A;
   function Get_A (This : Basic_Record) return Integer is
   begin
      return This.A;
   end Get_A;
end Package_With_Body;
```

pragma Pure_Function
Only available when using GNAT.

Two Flavors of Package

The packages above each define a type together with operations of the type. When the type's composition is placed in the private part of a package, the package then exports what is known to be an Abstract Data Type or ADT for short. Objects of the type are then constructed by calling one of the subprograms associated with the respective type.

A different kind of package is the Abstract State Machine. A package will be modelling a single item of the problem domain, such as the motor of a car. If a program controls one car, there is typically just one motor, or *the* motor. The public part of the package specification only declares the operations of the module (of the motor, say), but no type. All data of the module are hidden in the body of the package where they act as state variables to be queried, or manipulated by the subprograms of the package. The

initialization part sets the state variables to their initial values.

```
package Package_With_Body is

   procedure Set_A (An_A : in Integer);

   function Get_A return Integer;

private

   pragma Pure_Function (Get_A);

end Package_With_Body;


package body Package_With_Body is

   The_A: Integer;

   procedure Set_A (An_A : in Integer)
   is
   begin
      The_A := An_A;
   end Set_A;

   function Get_A return Integer is
   begin
      return The_A;
   end Get_A;


begin

   The_A := 0;

end Package_With_Body;
```

(A note on construction: The package initialization part after **begin** corresponds to a construction subprogram of an ADT package. However, as a state machine *is* an "object" already, "construction" is happening during package initialization. (Here it sets the state variable `The_A` to its initial value.) An ASM package can be viewed as a singleton.)

Using packages

This section is a stub.
You can help Wikibooks by expanding it.

To utilize a package it's needed to name it in a **with** clause, whereas to have direct visibility of that package it's needed to name it in a **use** clause.

For C++ programmers, Ada's **with** clause is analogous to the C++ preprocessor's **#include** and Ada's **use** is similar to the **using namespace** statement in C++. In particular, **use** leads to the same namespace pollution problems as **using namespace** and thus should be used sparingly. Renaming can shorten long compound names to a manageable length, while the **use type** clause makes a type's operators visible. These features reduce the need for plain **use**.

Standard with

The standard with clause provides visibility for the public part of a unit to the following defined unit.

The imported package can be used in any part of the defined unit, including the body when the clause is used in the specification.

Private with

This language feature is only available in Ada 2005 standard.

```
private with Ada.Strings.Unbounded;
package Private_With is

   -- The package Ada.String.Unbounded is not visible at this point

   type Basic_Record is private;
   procedure Set_A (This : in out Basic_Record;
                    An_A : in     String);
   function Get_A (This : Basic_Record) return String;
private
   -- The visibility of package Ada.String.Unbounded starts here

   package Unbounded renames Ada.Strings.Unbounded;
   type Basic_Record is
      record
         A : Unbounded.Unbounded_String;
      end record;
   pragma Pure_Function  (Get_A);
   pragma Inline (Get_A);
   pragma Inline (Set_A);
end Private_With;


package body Private_With is

   -- The private withed package is visible in the body too

   procedure Set_A (This : in out Basic_Record;
                    An_A : in     String)
   is
   begin
      This.A := Unbounded.To_Unbounded_String (An_A);
   end Set_A;
   function Get_A (This : Basic_Record) return String is
   begin
      return Unbounded.To_String (This.A);
   end Get_A;
end Private_With;
```

Limited with

This language feature is only available in Ada 2005 standard.

```
limited with Departments;
package Employees is

   type Employee is tagged private;
   procedure Assign_Employee
     (E : in out Employee;
      D : access Departments.Department'Class);
   type Dept_Ptr is access all Departments.Department'Class;
   function Current_Department(E : in Employee) return Dept_Ptr;
   ...
end Employees;


limited with Employees;
package Departments is

   type Department is tagged private;
   procedure Choose_Manager
     (Dept    : in out Department;
      Manager : access Employees.Employee'Class);
   ...
end Departments;
```

Making operators visible

Suppose you have a package Universe that defines some numeric type T.

```
with Universe;
procedure P is
  V: Universe.T := 10.0;
begin
  V := V * 42.0;  --  illegal
end P;
```

This program fragment is illegal since the operators implicitly defined in Universe are not directly visible.

You have four choices to make the program legal.

Use a use_package_clause. This makes **all declarations** in Universe directly visible (provided they are not hidden because of other homographs).

```
with Universe;
use  Universe;
procedure P is
  V: Universe.T := 10.0;
begin
  V := V * 42.0;
end P;
```

Use renaming. This is error prone since if you rename many operators, cut and paste errors are probable.

```
with Universe;
procedure P is
  function "*" (Left, Right: Universe.T) return Universe.T renames Universe."*";
  function "/" (Left, Right: Universe.T) return Universe.T renames Universe."*";  --  oops
  V: Universe.T := 10.0;
begin
  V := V * 42.0;
end P;
```

Use qualification. This is extremly ugly and unreadable.

```
with Universe;
procedure P is
  V: Universe.T := 10.0;
begin
  V := Universe."*" (V, 42.0);
end P;
```

Use the use_type_clause. This makes only the **operators** in Universe directly visible.

```
with Universe;
procedure P is
  V: Universe.T := 10.0;
  use type Universe.T;
begin
  V := V * 42.0;
end P;
```

There is a special beauty in the use_type_clause. Suppose you have a set of packages like so:

```
with Universe;
package Pack is
  subtype T is Universe.T;
end Pack;
```

```
with Pack;
procedure P is
  V: Pack.T := 10.0;
begin
  V := V * 42.0;  --  illegal
end P;
```

Now you've got into trouble. Since Universe is not made visible, you cannot use a use_package_clause for Universe to make the operator directly visible, nor can you use qualification for the same reason. Also a use_package_clause for Pack does not help, since the operator is not defined in Pack. The effect of the above construct means that the operator is not namable, i.e. it cannot be renamed in a renaming statement.

Of course you can add Universe to the context clause, but this may be impossible due to some other reasons (e.g. coding standards); also adding the operators to Pack may be forbidden or not feasible. So what to do?

The solution is simple. Use the use_type_clause for Pack.T and all is well!

```
with Pack;
procedure P is
  V: Pack.T := 10.0;
  use type Pack.T;
begin
  V := V * 42.0;
end P;
```

Package organisation

This section is a stub.
You can help Wikibooks by expanding it.

Nested packages

A nested package is a package declared inside a package. Like a normal package, it has a public part and a private part. From outside, items declared in a nested package N will have visibility as usual; the programmer may refer to these items using a full dotted name like `P.N.X`. (But not `P.M.Y`.)

```
package P is
   D: Integer;

   --  a nested package:
   package N is
      X: Integer;
   private
      Foo: Integer;
   end N;

   E: Integer;
private
   --  another nested package:
   package M is
      Y: Integer;
   private
      Bar: Integer;
   end M;

end P;
```

Inside a package, declarations become visible as they are introduced, in textual order. That is, a nested package N that is declared *after* some other declaration D can refer to this declaration D. A declaration E following N can refer to items of N[6]. But neither can "look ahead" and refer to any declaration that goes after them. For example, spec `N` above cannot refer to `M` in any way.

In the following example, a type is derived in both of the two nested packages `Disks` and `Books`. Notice that the full declaration of parent type `Item` appears before the two nested packges.

```
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;

package Shelf is

   pragma Elaborate_Body;

   --  things to put on the shelf

   type ID is range 1_000 .. 9_999;
   type Item (Identifier : ID) is abstract tagged limited null record;
   type Item_Ref is access constant Item'class;

   function Next_ID return ID;
   --  a fresh ID for an Item to Put on the shelf


   package Disks is

      type Music is (
         Jazz,
         Rock,
         Raga,
         Classic,
         Pop,
         Soul);

      type Disk (Style : Music; Identifier : ID) is new Item (Identifier)
         with record
            Artist : Unbounded_String;
            Title  : Unbounded_String;
         end record;

   end Disks;


   package Books is

      type Literature is (
         Play,
         Novel,
         Poem,
         Story,
         Text,
         Art);

      type Book (Kind : Literature; Identifier : ID) is new Item (Identifier)
         with record
            Authors : Unbounded_String;
            Title   : Unbounded_String;
            Year    : Integer;
         end record;

   end Books;

   --  shelf manipulation

   procedure Put (it: Item_Ref);
   function Get (identifier : ID) return Item_Ref;
   function Search (title : String) return ID;

private

   --  keeping private things private

   package Boxes is
      type Treasure(Identifier: ID) is limited private;
   private
      type Treasure(Identifier: ID) is new Item(Identifier) with null record;
   end Boxes;
```

```
end Shelf;
```

A package may also be nested inside a subprogram. In fact, packages can be declared in any declarative part, including those of a block.

Child packages

Ada allows one to extend the functionality of a unit (package) with so-called children (child packages). With certain exceptions, all the functionality of the parent is available to a child. This means that all public and private declarations of the parent package are visible to all child packages.

The above example, reworked as a hierarchy of packages, looks like this. Notice that the package Ada.Strings.Unbounded is not needed by the top level package `Shelf`, hence its with clause doesn't appear here. (We have added a match function for searching a shelf, though):

```
package Shelf is

   pragma Elaborate_Body;

   type ID is range 1_000 .. 9_999;
   type Item (Identifier : ID) is abstract tagged limited null record;
   type Item_Ref is access constant Item'Class;

   function Next_ID return ID;
   --  a fresh ID for an Item to Put on the shelf

   function match (it : Item; Text : String) return Boolean is abstract;
   --  see whether It has bibliographic information matching Text


   --   shelf manipulation

   procedure Put (it: Item_Ref);
   function Get (identifier : ID) return Item_Ref;
   function Search (title : String) return ID;

end Shelf;
```

The name of a child package consists of the parent unit's name followed by the child package's identifier, separated by a period (dot) `.'.

```
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;

package Shelf.Books is

   type Literature is (
      Play,
      Novel,
      Poem,
      Story,
      Text,
      Art);

   type Book (Kind : Literature; Identifier : ID) is new Item (Identifier)
      with record
         Authors : Unbounded_String;
         Title   : Unbounded_String;
         Year    : Integer;
      end record;

   function match(it: Book; text: String) return Boolean;

end Shelf.Books;
```

`Book` has two components of type `Unbounded_String`, so Ada.Strings.Unbounded appears in a

with clause of the child package. This is unlike the nested packages case which requires that all units needed by any one of the nested packages be listed in the context clause of the enclosing package (see 10.1.2 Context Clauses - With Clauses (Annotated)). Child packages thus give better control over package depenences. With clauses are more local.

The new child package `Shelf.Disks` looks similar. The `Boxes` package which was a nested package in the private part of the original `Shelf` package is moved to a private child package:

```
private package Shelf.Boxes is
    type Treasure(Identifier: ID) is limited private;
private
    type Treasure(Identifier: ID) is new Item(Identifier) with null record;
    function match(it: Treasure; text: String) return Boolean;
end Shelf.Boxes;
```

The privacy of the package means that it can only be used by equally private client units. These clients include private siblings and also the bodies of siblings (as bodies are never public).

Child packages may be listed in context clauses just like normal packages. A **with** of a child also 'withs' the parent.

Subunits

A subunit is just a feature to move a body into a place of its own when otherwise the enclosing body will become too large. It can also be used for limiting the scope of context clauses.

The subunits allow to physically divide a package into different compilation units without breaking the logical unity of the package. Usually each separated subunit goes to a different file allowing separate compilation of each subunit and independent version control history for each one.

```
package body Pack is
  procedure Proc is separate;
end Pack;

with Some_Unit;
separate (Pack)
procedure Proc is
begin
  ...
end Proc;
```

Notes

1. ↑ wikistats, Category:Ada Programming or /All Chapters)
2. ↑ paper by Eurocontrol (PDF, 160 kB) on Portability
3. ↑ Study by Stephen F. Zeigler on Ada vs. C
4. ↑ Main subprograms may even have parameters; it is implementation-defined what kinds of subprograms can be used as main subprograms. The reference manual explains the details in 10.2 LRM 10.2(29) (Annotated): "…, an implementation is required to support all main subprograms that are public parameterless library procedures." *Library* means not nested in another subprogram, for example, and other things that needn't concern us now.
5. ↑ **renames** can also be used for procedures, functions, variables, array elements. It can not be used for types - a type rename can be accomplished with **subtype**.
6. ↑ For example, `E: Integer := D + N.X;`

See also

Wikibook

- Ada Programming

Wikipedia

- Module

Ada 95 Reference Manual

- Section 7: Packages (Annotated)

Ada 2005 Reference Manual

- Section 7: Packages (Annotated)

18 Input Output

Ada has 5 independent libraries for input/output operations. So the most important lesson to learn is choosing the right one.

Text I/O

Text I/O is probably the most used Input/Output package. All data inside the file are represented by human readable text. Text I/O provides support for line and page layout but the standard is free form text.

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

procedure Main is
  Str  : String (1..5);
  Last : Natural;
begin
  Ada.Text_IO.Get_Line (Str, Last);
  Ada.Text_IO.Put_Line (Str (1..Last));
end;
```

Direct I/O

Direct I/O is used for random access files which contain only elements of one specific type. With Direct_IO you can position the file pointer to any element of that type (random access), however you can't freely choose the element type, the element type needs to be a definite subtype.

Sequential I/O

Direct I/O is used for random access files which contain only elements of one specific type. With Sequential_IO it is the other way round: you can choose between definite and indefinite element types but you have to read and write the elements one after the other.

Storage I/O

Storage I/O allows you to store *one* element inside a memory buffer. The element needs to be a definite subtype. Storage I/O is useful in Concurrent programming where it can be used to move elements from one task to another.

Stream I/O

Stream I/O is the most powerful input/output package which Ada provides. Stream I/O allows you to mix objects from different element types in one sequential file. In order to read/write from/to a stream each type provides a 'Read and 'Write attribute as well as an 'Input and 'Output attribute. These attributes are automatically generated for each type you declare.

The 'Read and 'Write attributes treat the elements as raw data. They are suitable for low level input/output as well as interfacing with other programming languages.

The 'Input and 'Output attribute add additional control informations to the file, like for example the 'First and 'Last attributes from an array.

In object orientated programming you can also use the 'Class'Input and 'Class'Output attributes - they will store and recover the actual object type as well.

Stream I/O is also the most flexible input/output package. All I/O attributes can be replaced with user defined functions or procedures using representation clauses and you can provide your own Stream I/O types using flexible object oriented techniques.

See also

Wikibook

- Ada Programming
- Ada Programming/Libraries/Ada.Direct_IO
- Ada Programming/Libraries/Ada.Sequential_IO
- Ada Programming/Libraries/Ada.Storage_IO
- Ada Programming/Libraries/Ada.Streams
  - Ada Programming/Libraries/Ada.Streams.Stream_IO
- Ada Programming/Libraries/Ada.Text_IO
  - Ada Programming/Libraries/Ada.Text_IO.Complex_IO (nested package)
- Ada Programming/Libraries/Ada.Text_IO.Decimal_IO (nested package)
- Ada Programming/Libraries/Ada.Text_IO.Enumeration_IO (nested package)
- Ada Programming/Libraries/Ada.Text_IO.Fixed_IO (nested package)
- Ada Programming/Libraries/Ada.Text_IO.Float_IO (nested package)
- Ada Programming/Libraries/Ada.Text_IO.Integer_IO (nested package)
- Ada Programming/Libraries/Ada.Text_IO.Modular_IO (nested package)
- Ada Programming/Libraries/Ada.Text_IO.Editing
- Ada Programming/Libraries/Ada.Float_Text_IO
- Ada Programming/Libraries/Ada.Integer_Text_IO

Ada 95 Reference Manual

- 13.13 Streams (Annotated)
- A.8.1 The Generic Package Sequential_IO (Annotated)
- A.8.4 The Generic Package Direct_IO (Annotated)
- A.10.1 The Package Text_IO (Annotated)
- A.12.1 The Package Streams.Stream_IO (Annotated)

Ada 2005 Reference Manual

- 13.13 Streams (Annotated)

- A.8.1 The Generic Package Sequential_IO (Annotated)
- A.8.4 The Generic Package Direct_IO (Annotated)
- A.10.1 The Package Text_IO (Annotated)
- A.12.1 The Package Streams.Stream_IO (Annotated)

Ada Quality and Style Guide

- 7.7 Input/Output
  - 7.7.1 Name and Form Parameters
- 7.7.2 File Closing
- 7.7.3 Input/Output on Access Types
- 7.7.4 Package Ada.Streams.Stream_IO
- 7.7.5 Current Error Files

19 EXCEPTIONS

Robustness

Robustness is the ability of a system or system component to behave "reasonably" when it detects an anomaly, e.g.:

- It receives invalid inputs.
- Another system component (hardware or software) malfunctions.

Take as example a telephone exchange control program. What should the control program do when a line fails? It is unacceptable simply to halt — all calls will then fail. Better would be to abandon the current call (only), record that the line is out of service, and continue. Better still would be to try to reuse the line — the fault might be transient. Robustness is desirable in all systems, but It is essential in systems on which human safety or welfare depends, e.g., hospital patient monitoring, aircraft fly-by-wire, nuclear power station control, etc.

Modules, preconditions and postconditions

A module may be specified in terms of its preconditions and postconditions. A *precondition* is a condition that the module's inputs are supposed to satisfy. A *postcondition* is a condition that the module's outputs are required to satisfy, provided that the precondition is satisfied. What should a module do if its precondition is not satisfied?

- Halt? Even with diagnostic information, this is generally unacceptable.
- Use a global result code? The result code can be set to indicate an anomaly. Subsequently it may be tested by a module that can effect error recovery. Problem: this induces tight coupling among the modules concerned
- Each module has its own result code? This is a parameter (or function result) that may be set to indicate an anomaly, and is tested by calling modules. Problems: (1) setting and testing result codes tends to swamp the normal-case logic and (2) the result codes are normally ignored.
- Exception handling — Ada's solution. A module detecting an anomaly raises an exception. The same, or another, module may handle that exception.

The exception mechanism permits clean, modular handling of anomalous situations:

- A unit (e.g., block or subprogram body) may raise an exception, to signal that an anomaly has been detected. The computation that raised the exception is abandoned (and can never be resumed, although it can be restarted).
- A unit may propagate an exception that has been raised by itself (or propagated out of another unit it has called).
- A unit may alternatively handle such an exception, allowing programmer-defined recovery from an anomalous situation. Exception handlers are segregated from normal-case code.

Predefined exceptions

The predefined exceptions are those defined in package Standard. Every language-defined run-time error causes a predefined exception to be raised. Some examples are:

- `Constraint_Error`, raised when a subtype's constraint is not satisfied
- `Program_Error`, when a protected operation is called inside a protected object, e.g.
- `Storage_Error`, raised by running out of storage
- `Tasking_Error`, when a task cannot be activated because the operating system has not enough resources, e.g.

Ex.1

```
Name : String (1 .. 10);
...
Name := "Hamlet"; -- Raises Constraint_Error,
                  -- because the "Hamlet" has bounds (1 .. 6).
```

Ex.2

```
loop
   P := new Int_Node'(0, P);
end loop; -- Soon raises Storage_Error,
          -- because of the extreme memory leak.
```

Ex.3 Compare the following approaches:

```
procedure Compute_Sqrt (X    : in  Float;
                        Sqrt : out Float;
                        OK   : out Boolean)
is
begin
   if X >= 0 then
      OK := True;
      -- compute √X
      ...
   else
      OK := False;
   end if;
end Compute_Sqrt;

...

procedure Triangle (A, B, C         : in  Float;
                    Area, Perimeter : out Float;
                    Exists          : out Boolean)
is
   S  : Float := 0.5 * (A + B + C);
   OK : Boolean;
begin
   Compute_Sqrt (S * (S-A) * (S-B) * (S-C), Area, OK);
   Perimeter := 2.0 * S;
   Exists    := OK;
end Triangle;
```

A negative argument to Compute_Sqrt causes OK to be set to False. Triangle uses it to determine its own status parameter value, and so on up the calling tree, *ad nauseam.*

versus

```
function Sqrt (X : Float) return Float is
begin
   if X < 0.0 then
      raise Constraint_Error;
   end if;
   -- compute √X
```

```
   ...
end Sqrt;

...

procedure Triangle (A, B, C         : in  Float;
                    Area, Perimeter : out Float)
is
   S  : Float := 0.5 * (A + B + C);
   OK : Boolean;
begin
   Area      := Sqrt (S * (S-A) * (S-B) * (S-C));
   Perimeter := 2.0 * S;
end Triangle;
```

A negative argument to Sqrt causes Constraint_Error to be explicitly raised inside Sqrt, and propagated out. Triangle simply propagates the exception (by not handling it).

Alternatively, we can catch the error by using the type system:

```
subtype Pos_Float is Float range 0.0 .. Float'Last;

function Sqrt (X : Pos_Float) return Pos_Float is
begin
   -- compute √X
   ...
end Sqrt;
```

A negative argument to Sqrt now raises Constraint_Error at the point of call. Sqrt is never even entered.

Input-output exceptions

Some examples of exceptions raised by subprograms of the **predefined package** Ada.Text_IO are:

- `End_Error`, raised by Get, Skip_Line, etc., if end-of-file already reached.
- `Data_Error`, raised by Get in Integer_IO, etc., if the input is not a literal of the expected type.
- `Mode_Error`, raised by trying to read from an output file, or write to an input file, etc
- `Layout_Error`, raised by specifying an invalid data format in a text I/O operation

Ex. 1

```
declare
   A : Matrix (1 .. M, 1 .. N);
begin
   for I in 1 .. M loop
      for J in 1 .. N loop
         begin
            Get (A(I,J));
         exception
            when Data_Error =>
               Put ("Ill-formed matrix element");
               A(I,J) := 0.0;
         end;
      end loop;
   end loop;
exception
   when End_Error =>
      Put ("Matrix element(s) missing");
end;
```

Exception declarations

Exceptions are declared rather like objects, but they are not objects. For example, recursive re-entry to a scope where an exception is declared does *not* create a new exception of the same name; instead the exception declared in the outer invocation is reused.

Ex.1

```
Line_Failed : exception;
```

Ex.2

```
package Directory_Enquiries is

   procedure Insert (New_Name    : in Name;
                     New_Number : in Number);
   procedure Lookup (Given_Name  : in  Name;
                     Corr_Number : out Number);
   Name_Duplicated : exception;
   Name_Absent     : exception;
   Directory_Full  : exception;
end Directory_Enquiries;
```

Raising exceptions

The **raise** statement explicitly raises a specified exception.

Ex. 1

```
package body Directory_Enquiries is

   procedure Insert (New_Name    : in Name;
                     New_Number : in Number)
   is
      …
   begin
      …
      if New_Name = Old_Entry.A_Name then
         raise Name_Duplicated;
      end if;
      …
      New_Entry :=  new Dir_Node'(New_Name, New_Number,…);
      …
   exception
      when Storage_Error => raise Directory_Full;
   end Insert;

   procedure Lookup (Given_Name  : in  Name;
                     Corr_Number : out Number)
   is
      …
   begin
      …
      if not Found then
         raise Name_Absent;
      end if;
      …
   end Lookup;

end Directory_Enquiries;
```

Exception handling and propagation

Exception handlers may be grouped at the end of a block, subprogram body, etc. A handler is any sequence of statements that may end:

- by completing;
- by executing a **return** statement;
- by raising a different exception (**raise** e;);
- by re-raising the same exception (**raise**;).

Suppose that an exception *e* is raised in a sequence of statements *U* (a block, subprogram body, etc.).

- If *U* contains a handler for *e*: that handler is executed, then control leaves *U*.
- If *U* contains no handler for *e*: *e* is *propagated* out of *U*; in effect, *e* is raised at the "point of call" of *U*.

So the raising of an exception causes the sequence of statements responsible to be abandoned at the point of occurrence of the exception. It is not, and cannot be, resumed.

Ex. 1

```
...
exception
   when Line_Failed =>
      begin -- attempt recovery
         Log_Error;
         Retransmit (Current_Packet);
      exception
         when Line_Failed =>
            Notify_Engineer; -- recovery failed!
            Abandon_Call;
      end;
...
```

Information about an exception occurrence

Ada provides information about an exception in an object of type Exception_Occurrence, defined in Ada.Exceptions along with subprograms taking this type as parameter:

- Exception_Name: return the full exception name using the dot notation and in uppercase letters. For example, `Queue.Overflow`.
- Exception_Message: return the exception message associated with the occurrence.
- Exception_Information: return a string including the exception name and the associated exception message.

For getting an exception occurrence object the following syntax is used:

```
with Ada.Exceptions;  use Ada.Exceptions;
...
exception
  when Error: High_Pressure | High_Temperature =>
    Put ("Exception: ");
    Put_Line (Exception_Name (Error));
    Put (Exception_Message (Error));
  when Error: others =>
    Put ("Unexpected exception: ");
    Put_Line (Exception_Information(Error));
end;
```

The exception message content is implementation defined when it is not set by the user who raises the exception. It usually contains a reason for the exception and the raising location.

The user can specify a message using the procedure Raise_Exception.

```
declare
   Valve_Failure : exception;
begin
  ...
  Raise_Exception (Valve_Failure'Identity, "Failure while opening");
  ...
  Raise_Exception (Valve_Failure'Identity, "Failure while closing");
  ...
exception
  when Fail: Valve_Failure =>
    Put (Exceptions_Message (Fail));
end;
```

The package also provides subprograms for saving exception occurrences and reraising them.

See also

Wikibook

- Ada Programming

Ada 95 Reference Manual

- Section 11: Exceptions (Annotated)
- 11.4.1 The Package Exceptions (Annotated)

Ada 2005 Reference Manual

- Section 11: Exceptions (Annotated)
- 11.4.1 The Package Exceptions (Annotated)

Ada Quality and Style Guide

- **Chapter 4: Program Structure**
  - 4.3 Exceptions
    - 4.3.1 Using Exceptions to Help Define an Abstraction
- **Chapter 5: Programming Practices**
  - 5.8 Using Exceptions
    - 5.8.1 Handling Versus Avoiding Exceptions
- 5.8.2 Handling for Others
- 5.8.3 Propagation
- 5.8.4 Localizing the Cause of an Exception

20 GENERICS

Parametric polymorphism (generic units)

The idea of code reusability arises because of the necessity to construct programs on the basis of well established building blocks that can be combined to form an ampler and complex system. The reusability of code improves the productivity and the quality of software. One of the ways in which the Ada language supports this characteristic is by means of generic units. A generic unit is a unit that defines algorithms in terms of types and other operations that are not defined until the user instantiates them. Generic units can be subprograms and packages.

Note to C++ programmers: generic units are similar to C++ templates.

For example, to define a procedure for swaping variables of any (non-limited) type:

```
generic
  type Element_T is private;  -- Generic formal type parameter
procedure Swap (X, Y : in out Element_T);


procedure Swap (X, Y : in out Element_T) is
  Temporary : constant Element_T := X;
begin
  X := Y;
  Y := Temporary;
end Swap;
```

The `Swap` subprogram is said to be generic. The subprogram specification is preceded by the generic formal part consisting of the reserved word generic followed by a list of generic formal parameters which may be empty. The entities declared as generic are not directly usable, it is necessary to instantiate them.

To be able to use `Swap`, it is necessary to create an instance for the wanted type. For example:

```
procedure Swap_Integers is new Swap (Integer);
```

Now the `Swap_Integers` procedure can be used for variables of type `Integer`.

The generic procedure can be instantiated for all the needed types. It can be instantiated with different names or, if the same identifier is used in the instantiation, each declaration overloads the procedure:

```
procedure Instance_Swap is new Swap (Float);
procedure Instance_Swap is new Swap (Day_T);
procedure Instance_Swap is new Swap (Element_T => Stack_T);
```

Similarly, generic packages can be used, for example, to implement a stack of any kind of elements:

```
generic
  Max: Positive;
  type Element_T is private;
package Generic_Stack is
  procedure Push (E: Element_T);
  function Pop return Element_T;
end Generic_Stack;


package body Generic_Stack is
  Stack: array (1 .. Max) of Element_T;
  Top  : Integer range 0 .. Max := 0;  -- initialise to empty
  -- ...
```

```
end Generic_Stack;
```

A stack of a given size and type could be defined in this way:

```
declare
  package Float_100_Stack is new Generic_Stack (100, Float);
  use Float_100_Stack;
begin
  Push (45.8);
  -- ...
end;
```

Generic parameters

The generic unit declares *generic formal parameters*, which can be:

- objects (of mode *in* or *in out* but never *out*)
- types
- subprograms
- instances of another, designated, generic unit.

When instantiating the generic, the programmer passes one *actual parameter* for each formal. Formal values and subprograms can have defaults, so passing an actual for them is optional.

Generic formal objects

Formal parameters of mode *in* accept any value, constant, or variable of the designated type. The actual is copied into the generic instance, and behaves as a constant inside the generic; this implies that the designated type cannot be limited. It is possible to specify a default value, like this:

```
generic
   Object : in Natural := 0;
```

For mode *in out*, the actual must be a variable.

One limitation with generic formal objects is that they are never considered static, even if the actual happens to be static. If the object is a number, it cannot be used to create a new type. It can however be used to create a new derived type, or a subtype:

```
generic
   Size : in Natural := 0;
package P is
   type T1 is mod Size; -- illegal!
   type T2 is range 1 .. Size; -- illegal!
   type T3 is new Integer range 1 .. Size; -- OK
   subtype T4 is Integer range 1 .. Size; -- OK
end P;
```

The reason why formal objects are nonstatic is to allow the compiler to emit the object code for the generic only once, and to have all instances share it, passing it the address of their actual object as a parameter. This bit of compiler technology is called *shared generics*. If formal objects were static, the compiler would have to emit one copy of the object code, with the object embedded in it, for each instance, potentially leading to an explosion in object code size (*code bloat*).

(Note to C++ programmers: in C++, since formal objects can be static, the compiler cannot

implement shared generics in the general case; it would have to examine the entire body of the generic before deciding whether or not to share its object code. In contrast, Ada generics are designed so that the compiler can instantiate a generic *without looking at its body*.)

Generic formal types

The syntax allows the programmer to specify which type classes are acceptable as actuals, as follows.

Generic formal type	Acceptable actual types
`type T (<>) is limited private;`	Any type at all. The actual type can be limited or not, but the *generic* treats it as limited, i.e. does not assume that assignment is available for the type. Rule of thumb: the syntax expresses *how the generic sees the type*, not *how the creator of the instance sees the type*.
`type T (<>) is private;`	Any nonlimited type: the generic knows that it is possible to assign to variables of this type.
`type T (<>) is abstract tagged limited private;`	Any tagged type, abstract or not, limited or not.
`type T (<>) is tagged limited private;`	Any concrete tagged type, limited or not.
`type T (<>) is abstract tagged private;`	Any nonlimited tagged type, abstract or not.
`type T (<>) is tagged private;`	Any nonlimited, concrete tagged type.
`type T (<>) is new Parent;`	Any type derived from `Parent`. The generic knows about `Parent`'s operations, so can call them. Neither `T` nor `Parent` can be abstract.
`type T (<>) is abstract new Parent with private;`	Any type, abstract or not, derived from `Parent`, where `Parent` is a tagged type, so calls to `T`'s operations can dispatch dynamically.
`type T (<>) is new Parent with private;`	Any concrete type, derived from the tagged type `Parent`.
`type T is (<>);`	Any discrete type: integer, modular, or enumeration.
`type T is range <>;`	Any signed integer type
`type T is delta <>;`	Any fixed point type
`type T is mod <>;`	Any modular type
`type T is digits <>;`	Any floating point type

`type T is delta <> digits <>;`	Any decimal fixed point type
`type T is array (I) of E;`	Any array type with index of type `I` and elements of type `E` (`I` and `E` could be formal parameters as well)
`type T is access O;`	Any access type pointing to objects of type `O` (`O` could be a formal parameter as well)

In the body we can only use the operations predefined for the type class of the formal parameter. That is, the generic specification is a contract between the generic implementor and the client instantiating the generic unit. This is different to the parametric features of other languages, such as C++.

In the table above, the first few type classes start with `type T (<>) is`.... This means that the generic sees T as having *unknown discriminants*. The actual may or may not have discriminants. It is possible to further restrict the set of acceptable actual types like so:

Generic formal type	Acceptable actual types
`type T (<>) is`...	Types with or without discriminants
`type T (D : DT) is`...	Types with a discriminant of type DT (it is possible to specify several discriminants, too)
`type T is`...	Types without a discriminant

Generic formal subprograms

It is possible to pass a subprogram as a parameter to a generic. The generic specifies a generic formal subprogram, complete with parameter list and return type (if the subprogram is a function). The actual must match this parameter profile. It is not necessary that the *names* of parameters match, though.

Here is the specification of a generic subprogram that takes another subprogram as its parameter:

```
generic
  type Element_T is private;
  with function "*" (X, Y: Element_T) return Element_T;
function Square (X : Element_T) return Element_T;
```

And here is the body of the generic subprogram; it calls parameter as it would any other subprogram.

```
function Square (X: Element_T) return Element_T is
begin
  return X * X;    -- The formal operator "*".
end Square;
```

This generic function could be used, for example, with matrices, having defined the matrix product.

```
with Square;
with Matrices;
procedure Matrix_Example is
  function Square_Matrix is new Square
    (Element_T => Matrices.Matrix_T, "*" => Matrices.Product);
  A: Matrix_T := Matrix_T.Identity;
begin
  A := Square_Matrix (A);
end Matrix_Example;
```

It is possible to specify a default with "the box" (`is <>`), like this:

```
generic
  type Element_T is private;
```

```
with function "*" (X, Y: Element_T) return Element_T is <>;
```

This means that if, at the point of instantiation, a function "*" exists for the actual type, and if it is directly visible, then it will be used by default as the actual subprogram.

One of the main uses is passing needed operators. The following example shows this (follow download links for full example):

File: Algorithms/binary_search.adb (view, plain text, download page, browse all)

```
generic
   type Element_Type is private;
   ...
   with function "<"
     (Left  : in Element_Type;
      Right : in Element_Type)
      return  Boolean
   is <>;
procedure Search
  (Elements : in Array_Type;
   Search   : in Element_Type;
   Found    : out Boolean;
   Index    : out Index_Type'Base)
   ...
```

Generic instances of other generic packages

A generic formal can be a package; it must be an instance of a generic package, so that the generic knows the interface exported by the package:

```
generic
   with package P is new Q (<>);
```

This means that the actual must be an instance of the generic package Q. The box after Q means that we do not care which actual generic parameters were used to create the actual for P. It is possible to specify the exact parameters, or to specify that the defaults must be used, like this:

```
generic
   -- P1 must be an instance of Q with the specified actual parameters:
   with package P1 is new Q (Param1 => X, Param2 => Y);
   -- P2 must be an instance of Q where the actuals are the defaults:
   with package P2 is new Q;
```

It is all or nothing: if you specify the generic parameters, you must specify all of them. Similarly, if you specify no parameters and no box, then all the generic formal parameters of Q must have defaults. The actual package must, of course, match these constraints.

The generic sees both the public part and the generic parameters of the actual package (Param1 and Param2 in the above example).

This feature allows the programmer to pass arbitrarily complex types as parameters to a generic unit, while retaining complete type safety and encapsulation. *(example needed)*

It is not possible for a package to list itself as a generic formal, so no generic recursion is possible. The following is illegal:

```
with A;
generic
   with package P is new A (<>);
```

```
package A; -- illegal: A references itself
```

In fact, this is only a particular case of:

```
with A; -- illegal: A does not exist yet at this point!
package A;
```

which is also illegal, despite the fact that A is no longer generic.

Instantiating generics

To instantiate a generic unit, use the keyword **new**:

```
function Square_Matrix is new Square
   (Element_T => Matrices.Matrix_T, "*" => Matrices.Product);
```

Notes of special interest to C++ programmers:

- The generic formal types define *completely* which types are acceptable as actuals; therefore, the compiler can instantiate generics without looking at the body of the generic.
- Each instance has a name and is different from all other instances. In particular, if a generic package declares a type, and you create two instances of the package, then you will get two different, incompatible types, even if the actual parameters are the same.
- Ada requires that all instantiations be explicit.
- It is not possible to create special-case instances of a generic (known as "template specialisation" in C++).

As a consequence of the above, Ada does not permit template metaprogramming. However, this design has significant advantages:

- the object code can be shared by all instances of a generic, unless of course the programmer has requested that subprograms be inlined; there is no danger of code bloat.
- when reading programs written by other people, there are no hidden instantiations, and no special cases to worry about. Ada follows the Law of Least Astonishment.

Advanced generics

Generics and nesting

A generic unit can be nested inside another unit, which itself may be generic. Even though no special rules apply (just the normal rules about generics and the rules about nested units), novices may be confused. It is important to understand the difference between a generic unit and *instances* of a generic unit.

Example 1. A generic subprogram nested in a nongeneric package.

```
package Bag_Of_Strings is
   type Bag is private;
   generic
      with procedure Operator (S : in out String);
```

```
   procedure Apply_To_All (B : in out Bag);
private
   -- omitted
end Bag_Of_Strings;
```

To use **Apply_To_All**, you first define the procedure to be applied to each String in the Bag. Then, you instantiate **Apply_To_All**, and finally you call the instance.

```
with Bag_Of_Strings;
procedure Example_1 is
   procedure Capitalize (S : in out String) is separate; -- omitted
   procedure Capitalize_All is
      new Bag_Of_Strings.Apply_To_All (Operator => Capitalize);
   B : Bag_Of_Strings.Bag;
begin
   Capitalize_All (B);
end Example_1;
```

Example 2. A generic subprogram nested in a generic package

This is the same as above, except that now the Bag itself is generic:

```
generic
   type Element_Type (<>) is private;
package Generic_Bag is
   type Bag is private;
   generic
      with procedure Operator (S : in out Element_Type);
   procedure Apply_To_All (B : in out Bag);
private
   -- omitted
end Generic_Bag;
```

As you can see, the generic formal subprogram **Operator** takes a parameter of the generic formal type **Element_Type**. This is okay: the nested generic seeseverything that is in its enclosing unit.

You cannot instantiate **Generic_Bag.Apply_To_All** directly, so you must first create an instance of **Generic_Bag**, say **Bag_Of_Strings**, and then instantiate **Bag_Of_Strings.Apply_To_All**.

```
with Generic_Bag;
procedure Example_2 is
   procedure Capitalize (S : in out String) is separate; -- omitted
   package Bag_Of_Strings is
      new Generic_Bag (Element_Type => String);
   procedure Capitalize_All is
      new Bag_Of_Strings.Apply_To_All (Operator => Capitalize);
   B : Bag_Of_Strings.Bag;
begin
   Capitalize_All (B);
end Example_2;
```

Generics and child units

Example 3. A generic unit that is a child of a nongeneric unit.

Each instance of the generic child is a child of the parent unit, and so it can see the parent's public and private parts.

```
package Bag_Of_Strings is
   type Bag is private;
private
   -- omitted
end Bag_Of_Strings;
generic
   with procedure Operator (S : in out String);
procedure Bag_Of_Strings.Apply_To_All (B : in out Bag);
```

The differences between this and Example 1 are:

- **Bag_Of_Strings.Apply_To_All** is compiled separately. In particular, **Bag_Of_Strings.Apply_To_All** might have been written by a different person who did not have access to the source text of **Bag_Of_Strings**.
- Before you can use **Bag_Of_Strings.Apply_To_All**, you must **with** it explicitly; **with**ing the parent, **Bag_Of_Strings**, is not sufficient.
- If you do not use **Bag_Of_Strings.Apply_To_All**, your program does not contain its object code.
- Because **Bag_Of_Strings.Apply_To_All** is at the library level, it can declare controlled types; the nested package could not do that in Ada 95. In Ada 2005, one can declare controlled types at any level.

```
with Bag_Of_Strings.Apply_To_All; -- implicitly withs Bag_Of_Strings, too
procedure Example_3 is
   procedure Capitalize (S : in out String) is separate; -- omitted
   procedure Capitalize_All is
      new Bag_Of_Strings.Apply_To_All (Operator => Capitalize);
   B : Bag_Of_Strings.Bag;
begin
   Capitalize_All (B);
end Example_3;
```

Example 4. A generic unit that is a child of a generic unit

This is the same as Example 3, except that now the Bag is generic, too.

```
generic
   type Element_Type (<>) is private;
package Generic_Bag is
   type Bag is private;
private
   -- omitted
end Generic_Bag;
generic
   with procedure Operator (S : in out Element_Type);
procedure Generic_Bag.Apply_To_All (B : in out Bag);
with Generic_Bag.Apply_To_All;
procedure Example_4 is
   procedure Capitalize (S : in out String) is separate; -- omitted
   package Bag_Of_Strings is
      new Generic_Bag (Element_Type => String);
   procedure Capitalize_All is
      new Bag_Of_Strings.Apply_To_All (Operator => Capitalize);
   B : Bag_Of_Strings.Bag;
begin
   Capitalize_All (B);
end Example_4;
```

Example 5. A parameterless generic child unit

Children of a generic unit **must** be generic, no matter what. If you think about it, it is quite logical: a child unit sees the public and private parts of its parent, including the variables declared in the parent. If the parent is generic, which instance should the child see? The answer is that the child must be the child of only one instance of the parent, therefore the child must also be generic.

```
generic
   type Element_Type (<>) is private;
   type Hash_Type is (<>);
   with function Hash_Function (E : Element_Type) return Hash_Type;
package Generic_Hash_Map is
   type Map is private;
private
   -- omitted
end Generic_Hash_Map;
```

Suppose we want a child of a **Generic_Hash_Map** that can serialise the map to disk; for this it needs to sort the map by hash value. This is easy to do, because we know that **Hash_Type** is a discrete type, and so has a less-than operator. The child unit that does the serialisation does not need any additional generic parameters, but it must be generic nevertheless, so it can see its parent's generic parameters, public and private part.

```
generic
package Generic_Hash_Map.Serializer is
    procedure Dump (Item : in Map; To_File : in String);
    procedure Restore (Item : out Map; From_File : in String);
end Generic_Hash_Map.Serializer;
```

To read and write a map to disk, you first create an instance of **Generic_Hash_Map**, for example **Map_Of_Unbounded_Strings**, and then an instance of **Map_Of_Unbounded_Strings.Serializer**:

```
with Ada.Strings.Unbounded;
with Generic_Hash_Map.Serializer;
procedure Example_5 is
   use Ada.Strings.Unbounded;
   function Hash (S : in Unbounded_String) return Integer is separate; -- omitted
   package Map_Of_Unbounded_Strings is
      new Generic_Hash_Map (Element_Type => Unbounded_String,
                            Hash_Type => Integer,
                            Hash_Function => Hash);
   package Serializer is
      new Map_Of_Unbounded_Strings.Serializer;
   M : Map_Of_Unbounded_Strings.Map;
begin
   Serializer.Restore (Item => M, From_File => "map.dat");
end Example_5;
```

See also

Wikibook

- Ada Programming
- Ada Programming/Object Orientation: tagged types provides other mean of polymorphism in Ada.

Wikipedia

- Generic programming

Ada Reference Manual

- Section 12: Generic Units (Annotated)

21 TASKING

Tasks

A *task unit* is a program unit that is obeyed concurrently with the rest of an Ada program. The corresponding activity, a new locus of control, is called a *task* in Ada terminology, and is similar to a *thread*, for example in Java Threads. The execution of the main program is also a task, the anonymous environment task. A task unit has both a declaration and a body, which is mandatory. A task body may be compiled separately as a subunit, but a task may not be a library unit, nor may it be generic. Every task depends on a *master*, which is the immediately surrounding declarative region - a block, a subprogram, another task, or a package. The execution of a master does not complete until all its dependant tasks have terminated. The environment task is the master of all the tasks declared in the program; it terminates only when they all have terminated.

Task units are similar to packages in that a task declaration defines entities exported from the task, whereas its body contains local declarations and statements of the task.

A single task is declared as follows:

```
task Single is
   declarations of exported identifiers
end Single;
...
task body Single is
   local declarations and statements
end Single;
```

A task declaration can be simplified, if nothing is exported, thus:

```
task No_Exports;
```

Ex. 1

```
procedure Housekeeping is

   task Check_CPU;
   task Backup_Disk;

   task body Check_CPU is
      ...
   end Check_CPU;

   task body Backup_Disk is
      ...
   end Backup_Disk;
   -- the two tasks are automatically created and begin execution
begin -- Housekeeping
   null;
   -- Housekeeping waits here for them to terminate
end Housekeeping;
```

It is possible to declare task types, thus allowing task units to be created dynamically, and incorporated in data structures:

```
task type T is
   ...
end T;
...
Task_1, Task_2 : T;
...
task body T is
   ...
```

```
end T;
```

Task types are **limited**, i.e. they are restricted in the same way as limited private types, so assignment and comparison are not allowed.

Rendezvous

The only entities that a task may export are entries. An **entry** looks much like a procedure. It has an identifier and may have **in**, **out** or **in out** parameters. Ada supports communication from task to task by means of the *entry call*. Information passes between tasks through the actual parameters of the entry call. We can encapsulate data structures within tasks and operate on them by means of entry calls, in a way analogous to the use of packages for encapsulating variables. The main difference is that an entry is executed by the called task, not the calling task, which is suspended until the call completes. If the called task is not ready to service a call on an entry, the calling task waits in a (FIFO) queue associated with the entry. This interaction between calling task and called task is known as a *rendezvous*. The calling task requests rendezvous with a specific named task by calling one of its entries. A task accepts rendezvous with any caller of a specific entry by executing an **accept** statement for the entry. If no caller is waiting, it is held up. Thus entry call and accept statement behave symmetrically. (To be honest, optimized object code may reduce the number of context switches below the number implied by this naive description.)

Ex. 2 The following task type implements a single-slot buffer, i.e. an encapsulated variable that can have values inserted and removed in strict alternation. Note that the buffer task has no need of state variables to implement the buffer protocol: the alternation of insertion and removal operations is directly enforced by the control structure in the body of Encapsulated_Buffer_Task_Type which is, as is typical, a **loop**.

```
task type Encapsulated_Buffer_Task_Type is
   entry Insert (An_Item : in  Item);
   entry Remove (An_Item : out Item);
end Encapsulated_Buffer_Task_Type;
...
Buffer_Pool : array (0 .. 15) of Encapsulated_Buffer_Task_Type;
This_Item   : Item;
...
task body Encapsulated_Buffer_Task_Type is
   Datum : Item;
begin
   loop
      accept Insert (An_Item : in  Item) do
         Datum := An_Item;
      end Insert;
      accept Remove (An_Item : out Item) do
         An_Item := Datum;
      end Remove;
   end loop;
end Encapsulated_Buffer_Task_Type;
...
Buffer_Pool(1).Remove (This_Item);
Buffer_Pool(2).Insert (This_Item);
```

Selective Wait

To avoid being held up when it could be doing productive work, a server task often needs the freedom to accept a call on any one of a number of alternative entries. It does this by means of the *selective wait* statement, which allows a task to wait for a call on any of two or more entries.

If only one of the alternatives in a selective wait statement has a pending entry call, then that one is accepted. If two or more alternatives have calls pending, the implementation is free to accept any one of

them. For example, it could choose one at random. This introduces *bounded non-determinism* into the program. A sound Ada program should not depend on a particular method being used to choose between pending entry calls. (However, there are facilities to influence the method used, when that is necessary.)

Ex. 3

```
task type Encapsulated_Variable_Task_Type is
   entry Store (An_Item : in  Item);
   entry Fetch (An_Item : out Item);
end Encapsulated_Variable_Task_Type;
...
task body Encapsulated_Variable_Task_Type is
   Datum : Item;
begin
   accept Store (An_Item : in Item) do
      Datum := An_Item;
   end Store;
   loop
      select
         accept Store (An_Item : in Item) do
            Datum := An_Item;
         end Store;
      or
         accept Fetch (An_Item : out Item) do
            An_Item := Datum;
         end Fetch;
      end select;
   end loop;
end Encapsulated_Variable_Task_Type;


x, y : Encapsulated_Variable_Task_Type;
```

creates two variables of type Encapsulated_Variable_Task_Type. They can be used thus:

```
it : Item;
...
x.Store(Some_Expression);
...
x.Fetch (it);
y.Store (it);
```

Again, note that the control structure of the body ensures that an Encapsulated_Variable_Task_Type must be given an initial value by a first Store operation before any Fetch operation can be accepted.

Guards

Depending on circumstances, a server task may not be able to accept calls for some of the entries that have accept alternatives in a selective wait statement. The acceptance of any alternative can be made conditional by using a *guard*, which is *Boolean* precondition for acceptance. This makes it easy to write monitor-like server tasks, with no need for an explicit signaling mechanism, nor for mutual exclusion. An alternative with a True guard is said to be *open*. It is an error if no alternative is open when the selective wait statement is executed, and this raises the Program_Error exception.

Ex. 4

```
task Cyclic_Buffer_Task_Type is
   entry Insert (An_Item : in  Item);
   entry Remove (An_Item : out Item);
end Cyclic_Buffer_Task_Type;
...
task body Cyclic_Buffer_Task_Type is
   Q_Size : constant := 100;
   subtype Q_Range is Positive range 1 .. Q_Size;
   Length : Natural range 0 .. Q_Size := 0;
   Head, Tail : Q_Range := 1;
```

```
   Data : array (Q_Range) of Item;
begin
   loop
      select
         when Length < Q_Size =>
            accept Insert (An_Item : in  Item) do
               Data(Tail) := An_Item;
            end Insert;
            Tail := Tail mod Q_Size + 1;
            Length := Length + 1;
      or
         when Length > 0 =>
            accept Remove (An_Item : out Item) do
               An_Item := Data(Head);
            end Remove;
            Head := Head mod Q_Size + 1;
            Length := Length - 1;
      end select;
   end loop;
end Cyclic_Buffer_Task_Type;
```

Protected types

Tasks allow for encapsulation and safe usage of variable data without the need for any explicit mutual exclusion and signaling mechanisms. Ex. 4 shows how easy it is to write server tasks that safely manage locally-declared data on behalf of multiple clients. There is no need for mutual exclusion of access to the managed data, *because it is never accessed concurrently*. However, the overhead of creating a task merely to serve up some data may be excessive. For such applications, Ada 95 provides **protected** modules. A protected module encapsulates a data structure and exports subprograms that operate on it under automatic mutual exclusion. It also provides automatic, implicit signaling of conditions between client tasks. Again, a protected module can be either a single protected object or a protected type, allowing many protected objects to be created.

A protected module can export only procedures, functions and entries, and its body may contain only the bodies of procedures, functions and entries. The protected data is declared after **private** in its specification, but is accessible only within the protected module's body. Protected procedures and entries may read and/or write its encapsulated data, and automatically exclude each other. Protected functions may only read the encapsulated data, so that multiple protected function calls can be concurrently executed in the same protected object, with complete safety; but protected procedure calls and entry calls exclude protected function calls, and vice versa. Exported entries and subprograms of a protected object are executed by its calling task, as a protected object has no independent locus of control. (To be honest, optimized object code may reduce the number of context switches below the number implied by this naive description.)

Like a task entry, a protected entry can employ a guard to control admission. This provides automatic signaling, and ensures that when a protected entry call is accepted, its guard condition is True, so that a guard provides a reliable precondition for the entry body.

Ex. 5 The following is a simple protected type analogous to the Encapsulated_Buffer task in Ex. 2.

```
protected type Protected_Buffer_Type is
   entry Insert (An_Item : in  Item);
   entry Remove (An_Item : out Item);
private
   Buffer : Item;
   Empty  : Boolean := True;
end Protected_Buffer_Type;
...
protected body Protected_Buffer_Type is
   entry Insert (An_Item : in  Item)
      when Empty is
   begin
```

```
      Buffer := An_Item;
      Empty := False;
   end Insert;
   entry Remove (An_Item : out Item)
      when not Empty is
   begin
      An_Item := Buffer;
      Empty := True;
   end Remove;
end Protected_Buffer_Type;
```

Note how the guards, using the state variable Empty, ensure that messages are alternately inserted and removed, and that no attempt can be made to take data from an empty buffer. All this is achieved without explicit signaling or mutual exclusion constructs, whether in the calling task or in the protected type itself.

The notation for calling a protected entry or procedure is exactly the same as that for calling a task entry. This makes it easy to replace one implementation of the abstract type by the other, the calling code being unaffected.

Ex. 6 The following task type implements Dijkstra's semaphore ADT, with FIFO scheduling of resumed processes. The algorithm will accept calls to both Wait and Signal, so long as the semaphore invariant would not be violated. When that circumstance approaches, calls to Wait are ignored for the time being.

```
task type Semaphore_Task_Type is
   entry Initialize (N : in Natural);
   entry Wait;
   entry Signal;
end Semaphore_Task_Type;
...
task body Semaphore_Task_Type is
   Count : Natural;
begin
   accept Initialize (N : in Natural) do
      Count := N;
   end Initialize;
   loop
      select
         when Count > 0 =>
             accept Wait do
                Count := Count - 1;
             end Wait;
      or
             accept Signal;
             Count := Count + 1;
      end select;
   end loop;
end Semaphore_Task_Type;
```

This task could be used as follows:

```
nr_Full, nr_Free : Semaphore_Task_Type;
...
nr_Full.Initialize (0); nr_Free.Initialize (nr_Slots);
...
nr_Free.Wait; nr_Full.Signal;
```

Alternatively, semaphore functionality can be provided by a protected object, with major efficiency gains.

Ex. 7 The Initialize and Signal operations of this protected type are unconditional, so they are implemented as protected procedures, but the Wait operation must be guarded and is therefore implemented as an entry.

```
protected type Semaphore_Protected_Type is
   procedure Initialize (N : in Natural);
```

```
   entry Wait;
   procedure Signal;
private
   Count : Natural := 0;
end Semaphore_Protected_Type;
...
protected body Semaphore_Protected_Type is
   procedure Initialize (N : in Natural) is
   begin
      Count := N;
   end Initialize;
   entry Wait
      when Count > 0 is
   begin
      Count := Count - 1;
   end Wait;
   procedure Signal is
   begin
      Count := Count + 1;
   end Signal;
end Semaphore_Protected_Type;
```

Unlike the task type above, this does not ensure that Initialize is called before Wait or Signal, and Count is given a default initial value instead. Restoring this defensive feature of the task version is left as an exercise for the reader.

Entry families

Sometimes we need a group of related entries. Entry *families*, indexed by a *discrete type*, meet this need.

Ex. 8 This task provides a pool of several buffers.

```
type Buffer_Id is Integer range 1 .. nr_Bufs;
...
task Buffer_Pool_Task is
   entry Insert (Buffer_Id) (An_Item : in Item);
   entry Remove (Buffer_Id) (An_Item : out Item);
end Buffer_Pool_Task;
...
task body Buffer_Pool_Task is
   Data   : array (Buffer_Id) of Item;
   Filled : array (Buffer_Id) of Boolean  := (others => False);
begin
  loop
    for I in Data'Range loop
      select
        when not Filled(I) =>
           accept Insert (I) (An_Item : in Item) do
              Data(I) := An_Item;
           end Insert;
           Filled(I) := True;
      or
        when Filled(I) =>
           accept Remove (I) (An_Item : out Item) do
                  An_Item := Data(I);
           end Remove;
           Filled(I) := False;
      else
        null; -- N.B. "polling" or "busy waiting"
      end select;
    end loop;
  end loop;
end Buffer_Pool_Task;
...
Buffer_Pool_Task.Remove(K)(This_Item);
```

Note that the busy wait **else null** is necessary here to prevent the task from being suspended on some buffer I when there was no call pending for it, because such suspension would delay serving requests for all the other buffers (perhaps indefinitely).

Termination

Server tasks often contain infinite loops to allow them to service an arbitrary number of calls in succession. But control cannot leave a task's master until the task terminates, so we need a way for a server to know when it should terminate. This is done by a *terminate alternative* in a selective wait.

Ex. 9

```
task type Terminating_Buffer_Task_Type is
   entry Insert (An_Item : in  Item);
   entry Remove (An_Item : out Item);
end Terminating_Buffer_Task_Type;
...
task body Terminating_Buffer_Task_Type is
   Datum : Item;
begin
   loop
      select
         accept Insert (An_Item : in  Item) do
            Datum := An_Item;
         end Insert;
      or
         terminate;
      end select;
      select
         accept Remove (An_Item : out Item) do
            An_Item := Datum;
         end Remove;
      or
         terminate;
      end select;
   end loop;
end Terminating_Buffer_Task_Type;
```

The task terminates when:

1. at least one terminate alternative is open, and
2. there are no pending calls to its entries, and
3. all other tasks of the same master are in the same state (or already terminated), and
4. the task's master has completed (i.e. has run out of statements to execute).

Conditions (1) and (2) ensure that the task is in a fit state to stop. Conditions (3) and (4) ensure that stopping cannot have an adverse effect on the rest of the program, because no further calls that might change its state are possible.

Timeout

A task may need to avoid being held up by calling to a slow server. A *timed entry call* lets a client specify a maximum delay before achieving rendezvous, failing which the attempted entry call is withdrawn and an alternative sequence of statements is executed.

Ex. 10

```
task Password_Server is
   entry Check (User, Pass : in String; Valid : out Boolean);
   entry Set (User, Pass : in  String);
end Password_Server;
...
User_Name, Password : String (1 .. 8);
...
Put ("Please give your new password:");
Get_Line (Password);
```

```
select
   Password_Server.Set (User_Name, Password);
   Put_Line ("Done");
or
   delay 10.0;
   Put_Line ("The system is busy now, please try again later.");
end select;
```

To time out the *functionality* provided by a task, two distinct entries are needed: one to pass in arguments, and one to collect the result. Timing out on rendezvous with the latter achieves the desired effect.

Ex. 11

```
task Process_Data is
   entry Input (D  : in  Datum);
   entry Output (D  : out Datum);
end Process_Data;

Input_Data, Output_Data : Datum;

loop
   collect Input_Data from sensors;
   Process_Data.Input (Input_Data);
   select
      Process_Data.Output (Output_Data);
      pass Output_Data to display task;
   or
      delay 0.1;
      Log_Error ("Processing did not complete quickly enough.");
   end select;
end loop;
```

Symmetrically, a delay alternative in a selective wait statement allows a server task to withdraw an offer to accept calls after a maximum delay in achieving rendezvous with any client.

Ex. 12

```
task Resource_Lender is
   entry Get_Loan (Period : in Duration);
   entry Give_Back;
end Resource_Lender;
...
task body Resource_Lender is
   Period_Of_Loan : Duration;
begin
   loop
      select
         accept Get_Loan (Period : in Duration) do
            Period_Of_Loan := Period;
         end Get_Loan;
         select
            accept Give_Back;
         or
            delay Period_Of_Loan;
            Log_Error ("Borrower did not give up loan soon enough.");
         end select;
      or
         terminate;
      end select;
   end loop;
end Resource_Lender;
```

Conditional entry calls

An entry call can be made conditional, so that it is withdrawn if rendezvous is not immediately achieved. This uses the select statement notation with an **else** part. The constructs:

```
select
```

```
   ...
else
   ...
end select;
```

and

```
select
   ...
or
   delay 0.0
   ...
end select;
```

are equivalent.

Requeue statements

A requeue statement allows an accept statement or entry body to be completed while redirecting to a different or the same entry queue. The called entry has to share the same parameter list or be parameterless.

Scheduling

FIFO, priority, priority inversion avoidance, ... to be completed

Interfaces

This language feature is only available in Ada 2005 standard.

Task and Protected types can also implement interfaces.

```
type Printable is task interface;
procedure Input (D  : in  Printable);
task Process_Data is new Printable with
   entry Input  (D  : in  Datum);
   entry Output (D  : out Datum);
end Process_Data;
```

See also

Wikibook

- Ada Programming

Ada Reference Manual

Ada 95

- Section 9: Tasks and Synchronization (Annotated)

Ada 2005

- 3.9.4 Interface Types (Annotated)
- Section 9: Tasks and Synchronization (Annotated)

Ada Quality and Style Guide

- **Chapter 4: Program Structure**
  - 4.1.9 Tasks
- 4.1.10 Protected Types
- Chapter 6: Concurrency

22 Object Orientation

Object-orientation on Ada

An Ada class consists of three building blocks:

- a package
- a record (data storage)
- primitive operations (methods).

This is different from C++ which has no concept ofa package.

Note: Rigorously the meaning of the term *class* is different in Ada than in C++, Java and others. In Ada, a class is a special type whose values are those of the union of all the types derived from a given tagged type, called the class-wide type. In C++, the term class is a bit polysemic, it does mean the former concept, and it is a concrete type with operations associated, and finally a construct to define them.

The *package*

Every class has to be defined inside a package. One package may contain more than one class. You might be surprised about that requirement but you should remember that every class needs some housekeeping information which has to be kept somewhere; in Ada this housekeeping information is part of the package.

A little note for C++ programmers

If you have ever, when linking a C++ program, got an error message telling you that the virtual function table for class X was not found then you might appreciate this restriction. In Ada you never get this error message - the Ada equivalent of the virtual function table is part of the package. The same goes for the Ada equivalent of a *virtual inline function*, another cause of trouble when linking C++ programs - and they too work flawlessly in Ada.

The *tagged record*

A tagged type provides support for dynamic polymorphism and type extension in Ada. A tagged type bears a hidden tag that identifies the type and its position in the inheritance tree at run-time. It also provides the means of storing instance data.

```
package Person is

   type Object is tagged
      record
         Name   : String (1 .. 10);
         Gender : Gender_Type;
      end record;
end Person;


with Person;
package Programmer is

   type Object is new Person.Object with
```

```
      record
         Skilled_In : Language_List;
      end record;
end Programmer;
```

Creating Objects

The first paragraph of the Language Reference Manual's section on 3.3 Objects and Named Numbers (Annotated) states when an object is created, and destroyed again. This subsection illustrates how objects are created.

In order to have an example, assume a typical hierarchy of object oriented types: a top-level type `Person`, a `Programmer` type derived from `Person`, and so on. Each person has a name, so assume `Person` objects to have a `Name` component. Likewise, he or she has a `Gender` component. The `Programmer` type inherits the components and the operations of the `Person` type, so `Programmer` objects have a `Name`, too, etc.

The LRM paragraph starts,

> *Objects are created at run time and contain a value of a given type. An object can be created and initialized as part of elaborating a declaration, evaluating an allocator, aggregate, or function_call.*

For a start, objects of a tagged type are created the same way as objects of any type. The second sentence means, for example, that an object will be created when you declare a variable or a constant of a type. For the tagged type `Person`,

```
declare
   P: Person;
begin
   Text_IO.Put_Line("The name is " & P.Name);
end;
```

Nothing special so far. Just like any ordinary variable declaration this O-O one is elaborated. The result of elaboration is an object named `P`, of type `Person`. However, `P` has only default name and gender value components. These are likely not useful ones. If we want the object to be initialized immediately we can provide an initial aggregate for `P`'s `Name` and `Gender` components.

```
declare
   P: Person := (Name => "Scorcese  ", Gender => Male);
begin
   Text_IO.Put_Line("The name is " & P.Name);
end;
```

The parenthesized expression after := is called an *aggregate* (4.3 Aggregates (Annotated)).

Another way to create an object that is mentioned in the LRM paragraph is to call a function. An object will be created as the return value of a function call. Introducing proper O-O information hiding, we change the package containing the `Person` type so that `Person` becomes a private type. To enable clients of the package to construct `Person` objects we declare a function that returns them. (The function may do some interesting construction work on the objects.) We also declare a function that returns the name of `Person` objects.

```
package Persons is

   type Person is tagged private;

   function Make (Name: String; Sex: Gender_Type) return Person;
```

```
      function Name (P: Person) return String;

   private
      type Person is tagged
         record
            Name   : String (1 .. 10);
            Gender : Gender_Type;
         end record;

   end Persons;
```

Calling the `Make` function results in another object which can be used for initialization, or for copying one object into another. We can no longer refer to the `Name` component of `P` since the `Person` type is **private**. But we have a corresponding function `Name` declared with the `Person` type. This is a socalled primitive operation. (The component and the function in this example are both named `Name` but this is accidental. We can choose a different name for either if we want.)

```
   declare
      P: Person := Make (Name => "Orwell    ", Sex => Male);
   begin
      if 2001 > 1984 then
         P := Make (Name => "Kubrick  ", Sex => Male);
      end if;

      Text_IO.Put_Line("The name is " & Name(P));
   end;
```

So far there is no mention of the `Programmer` type derived from `Person`. Before dealing with `Programmer` objects, thus introducing polymorphism and initialization in the presence of inheritance, a few words about class-wide types are in order.

The class-wide type

Whenever you declare a *tagged type*, the compiler will also provide a Class type for you. This type is called a *class-wide type*. Beginners often consider the class-wide type as some abstract concept. But this is not true! The Class type is a real data type. You can declare variables of the class-wide type, assign values to them, use the class-wide type as a parameter to functions and procedures and so forth.

The interesting part is that the class-wide type can hold values of any child class of the declared base type; in fact, there are no objects of this class-wide type - an object always is of a specific child type. This means that, unlike with most other object-oriented programming languages, you do not need to resort to the use of heap memory for storing arbitrary members of a class family but you can use the Class type instead.

Since the size of the child class is not known at compile time, any class-wide type is an indefinite subtype.

Primitive operations

The primitive operations of a given tagged type are those having a parameter or return value of this type and are declared immediately in the samepackage as the type.

Primitive operations are inherited and can be overridden for derived types.

Examples:

```
package X is

   type Object is tagged null record;
   procedure Class_Member_1 (This : in     Object);
   procedure Class_Member_2 (This :    out Object);
   procedure Class_Member_3 (This : in out Object);
   procedure Class_Member_4 (This : access Object);
   function  Class_Member_5 return Object;
end X;
```

A note for C++ programers: Class types are "by reference types" - they are always passed to a subprogram by using a pointer and never use "call by value" - using an access explicitly is therefore seldom needed.

Please note that neither named access types nor class-wide types qualify as class members. Procedures and functions having those as a parameter are just normal subprograms sharing the same package without belonging to the *class*.

Examples:

```
package X is

   type Object is tagged null record;
   type Object_Access is access Object;
   type Object_Class_Access is access Object'Class;
   procedure No_Class_Member_1 (This : in     Object'Class);
   procedure No_Class_Member_2 (This : in out Object_Access);
   procedure No_Class_Member_3 (This :    out Object_Class_Access);
   function  No_Class_Member_4 return Object'Class;
   package Inner is

      procedure No_Class_Member_5 (This : in Object);
   end Inner;
end X;
```

Note for C++ programmers: Procedures and functions like these can serve the same purpose as "non virtual" and "static" methods have in C++.

Ada 2005 adds a number of new features to the support for object oriented programming in Ada. Among these are overriding indicators and anonymous accesstypes.

This language feature is only available in Ada 2005 standard.

Anonymous access types can now be used at more places. So also a function like Class_Member_4 below is a primitive operation.

The new keyword **overriding** can be used to indicate whether an operation is overriding an inherited subprogram or not. Note that its use is optional because of upward-compatibility with Ada 95. For example:

```
package X is

   type Object is tagged null record;
   procedure Class_Member_1 (This : in     Object);
   procedure Class_Member_2 (This : in out Object);
   function  Class_Member_3 return Object;
   function  Class_Member_4 return access Object;
   type Derived_Object is new Object with null record;
   overriding
   procedure Class_Member_1 (This : in     Derived_Object);
   -- 'Class_Member_2' is not overridden

   overriding
   function  Class_Member_3 return Derived_Object;
   not overriding
   function  New_Class_Member_1 return Derived_Object;
```

```
end X;
```

The compiler will check the desired behaviour.

This is a good programming practice because it avoids some nasty bugs like not overriding an inherited subprogram because the programmer spelt the identifier incorrectly, or because a new parameter is added later in the parent type.

It can also be used with abstract operations, with renamings, or when instantiating a generic subprogram:

```
not overriding
procedure Class_Member_x (This : in Object) is abstract;

overriding
function  Class_Member_y return Object renames Old_Class_Member;
not overriding
procedure Class_Member_z (This : out Object)
      is new Generic_Procedure (Element => Integer);
```

The class-wide type, again

Class-wide types play an important part in dynamically dispatching calls. Primitive operations of a type are bound to a specific type. Each object carries with it a tag that identifies its specific type at run time, hence it identifies specific primitive operations. How, then, are operations selected for polymorphic entities? This is where class-wide types enter.

Consider the types Person.Object and Programmer.Object as above, with one primitive operation added to them:

```
package Person is
  type Object is tagged …
  procedure Inform (Who: in out Object; What: in String);

package Programmer is
  type Object is new Person.Object with …
  procedure Inform (Who: in out Object; What: in String);
```

Supposedly, each type of person needs to be informed in their own ways, so Programmer.Inform overrides Person.Inform. Now another subprogram is used to call operation `Inform` for objects of any Person type, that is, of any type in the class rooted at Person.

```
procedure Ask
   (Audience : in out Person.Object'Class;
    What : in String) is
begin
   Inform (Audience, What);
end Ask;
```

The effect of 'Class is that at run time, the specific object passed in for the parameter Audience will be inspected. It must be of type Person or of some type derived from Person. Depending on its run time tag, the proper `Inform` is selected for dispatching.

```
Coder: Programmer.Object;
Sales: Person.Object;

…

Ask (Sales, "Will you join us?");
--  dispatches to Person.Inform
```

```
Ask (Coder, "Do you want Pizza or Chop Soey?");
--  dispatches to Programmer.Inform
```

Abstract types

A tagged type can also be abstract (and thus can have abstract operations):

```
package X is

   type Object is abstract tagged …;
   procedure One_Class_Member      (This : in     Object);
   procedure Another_Class_Member  (This : in out Object);
   function  Abstract_Class_Member return Object  is abstract;
end X;
```

An abstract operation cannot have any body, so derived types are forced to override it (unless those derived types are also abstract). See next section about interfaces for more information about this.

The difference with a non-abstract tagged type is that you cannot declare any variable of this type. However, you can declare an access to it, and use it as a parameter of a class-wide operation.

Interfaces

This language feature is only available in Ada 2005 standard.

Interfaces allow for a limited form of multiple inheritance. On a sematic level they are similar to an "abstract tagged null record" as they may have primitive operations but cannot hold any data. These operations cannot have a body, so they are either declared **abstract** or **null**. *Abstract* means the operation has to be overridden, *null* means the default implementation is a null body, i.e. one that does nothing.

An interface is declared with:

```
package Printable is

   type Object is interface;
   procedure Class_Member_1 (This : in     Object) is abstract;
   procedure Class_Member_2 (This :    out Object) is null;
end Printable;
```

You implement an **interface** by adding it to a concrete *class*:

```
with Person;
package Programmer is

   type Object is new Person.Object
                  and Printable.Object
   with
      record
         Skilled_In : Language_List;
      end record;
   overriding
   procedure Class_Member_1   (This : in Object);
   not overriding
   procedure New_Class_Member (This : Object; That : String);
end Programmer;
```

As usual, all inherited abstract operations must be overridden although *null subprograms* ones need not.

Class names

Both the class package and the class record need a name. In theory they may have the same name, but in practice this leads to nasty (because of unintutive error messages) name clashes when you use the `use` clause. So over time three de facto naming standards have been commonly used.

Classes/Class

The package is named by a plural noun and the record is named by the corresponding singular form.

```
package Persons is

   type Person is tagged
      record
         Name   : String (1 .. 10);
         Gender : Gender_Type;
      end record;
end Persons;
```

This convention is the usually used in Ada's built-in libraries.

Disadvantage: Some "multiples" are tricky to spell, especially for those of us who aren't native English speakers.

Class/Object

The package is named after the class, the record is just named Object.

```
package Person is

   type Object is tagged
      record
         Name   : String (1 .. 10);
         Gender : Gender_Type;
      end record;
end Person;
```

Most UML and IDL code generators use this technique.

Disadvantage: You can't use the `use` clause on more than one such class packages at any one time. However you can always use the "type" instead of the package.

Class/Class_Type

The package is named after the class, the record is postfixed with *_Type*.

```
package Person is

   type Person_Type is tagged
      record
         Name   : String (1 .. 10);
         Gender : Gender_Type;
      end record;
end Person;
```

Disadvantage: lots of ugly "_Type" postfixes.

See also

Wikibook

- Ada Programming
- Ada Programming/Types/record
- record
- interface
- tagged

Wikipedia

- Object-oriented programming

Ada Reference Manual

Ada 95

- 3.8 Record Types (Annotated)
- 3.9 Tagged Types and Type Extensions (Annotated)
- 3.9.1 Type Extensions (Annotated)
- 3.9.2 Dispatching Operations of Tagged Types (Annotated)
- 3.9.3 Abstract Types and Subprograms (Annotated)
- 3.10 Access Types (Annotated)

Ada 2005

- 3.8 Record Types (Annotated)
- 3.9 Tagged Types and Type Extensions (Annotated)
- 3.9.1 Type Extensions (Annotated)
- 3.9.2 Dispatching Operations of Tagged Types (Annotated)
- 3.9.3 Abstract Types and Subprograms (Annotated)
- 3.9.4 Interface Types (Annotated)
- 3.10 Access Types (Annotated)

Ada Quality and Style Guide

- Chapter 9: Object-Oriented Features

23 Ada 2005

This is an overview of the major features that are available in the new Ada standard (accepted by ISO in January 2007; to differentiate it from its predecessors Ada 83 and Ada 95, the informal name Ada 2005 is generally agreed on). For the rationale and a more detailed (and very technical) description, see the draft of the Amendment to the Ada Reference Manual following the links to the last version of every **Ada Issue** document (AI).

Although the standard is now published, not all compilers will be able to handle it. Many of these additions are already implemented by the following Free Software compilers:

- GNAT GPL Edition
- GCC 4.1
- GNAT Pro 6.0.2 (the AdaCore supported version) is a complete implementation.

After downloading and installing any of them, remember to use the `-gnat05` switch when compiling Ada 2005 code. Note that in GNAT GPL 2007 Edition Ada 2005 is the default mode.

Language features

Character set

Not only does Ada 2005 now support a new 32-bit character type — called `Wide_Wide_Character` — but the source code itself may be of this extended character set as well. Thus Russians and Indians, for example, will be able to use their native language in identifiers and comments. And mathematicians will rejoice: The whole Greek and fractur character sets are available for identifiers. For example, Ada.Numerics will be extended with a new constant:

```
π : constant := Pi;
```

This is not a new idea — GNAT always had the `-gnatic` compiler option to specify the character set [3]. But now this idea has become standard, so all Ada compilers will need to support Unicode 4.0 for identifiers — as the new standard requires.

See also:

- AI-00285 Support for 16-bit and 32-bit characters
- AI-00388 Add Greek pi to Ada.Numerics

Interfaces

Interfaces allow for a limited form of multiple inheritance similar to Java and C#.

You find a full description here: Ada Programming/OO.

See also:

- AI-00251 Abstract Interfaces to provide multiple inheritance
- AI-00345 Protected and task interfaces

Union

In addition to Ada's safe variant record an unchecked C style union is now available.

You can find a full description here: Ada Programming/Types/record#Union.

See also:

- AI-00216 Unchecked unions -- variant records with no run-time discriminant
- Annex B.3.3 Pragma Unchecked_Union (Annotated)

With

The with statement got a massive upgrade. First there is the new limited with which allows two packages to *with* each other. Then there is private with to make a package only visible inside the private part of the specification.

See also:

- AI-50217 Limited With Clauses
- AI-00262 Access to private units in the private part

Access types

Not null access

An access type definition can specify that the accesstype can never be null.

See Ada Programming/Types/access#Not null access.

See also: AI-00231 Access-to-constant parameters and null-excluding access subtypes

Anonymous access

The possible uses of anonymous access types are extended. They are allowed virtually in every type or object definition, including access to subprogram parameters. Anonymous access types may point to constant objects as well. Also, they could be declared to be not null.

With the addition of the following operations in package Standard, it is possible to test the equality of anonymous access types.

```
function "=" (Left, Right : universal_access) return Boolean;
function "/="(Left, Right : universal_access) return Boolean;
```

See Ada Programming/Types/access#Anonymous access.

See also:

- AI-00230 Generalized use of anonymous access types
- AI-00385 Stand-alone objects of anonymous access types
- AI-10318 Limited and anonymous access return types

Language library

Containers

A major addition to the language library is the generic packages for containers. If you are familiar with the C++ STL, you will likely feel very much at home using Ada.Containers. One thing, though: Ada is a block structured language. Many ideas of how to use the STL employ this feature of the language. For example, local subprograms can be supplied to iteration schemes.

The original Ada Issue text AI-20302 Container library has now been transformed into A.18 Containers (Annotated).

If you know how to write Ada programs, and have a need for vectors, lists, sets, or maps (tables), please have a look at the AI-20302 AI Text mentioned above. There is an *!example* section in the text explaining the use of the containers in some detail. Matthew Heaney provides a number of demonstration programs with his reference implementation of AI-302 (Ada.Containers) which you can find at tigris.

In Ada Programming/Containers you will find a demo using containers.

Historical side note: The C++ STL draws upon the work of David R. Musser and Alexander A. Stepanov. For some of their studies of generic programming, they had been using Ada 83. The Stepanov Papers Collection has a few publications available.

Scan Filesystem Directories and Environment Variables

See also:

- AI-00248 Directory Operations
- AI-00370 Environment variables

Numerics

Besides the new constant of package Ada.Numerics (see Character Set above), the most important addition are the packages to operate with vectors and matrices.

See also:

- AI-00388 Add Greek pi (π) to Ada.Numerics

- AI-00296 Vector and matrix operations

(Related note on Ada programming tools: AI-388 contains an interesting assessment of how compiler writers are bound to perpetuate the lack of handling of international characters in programming support tools for now. As an author of Ada programs, be aware that your tools provider or Ada consultant could recommend that the program text be 7bit ASCII only.)

Real-Time and High Integrity Systems

See also:

- AI-00297 Timing events
- AI-00307 Execution-Time Clocks
- AI-00354 Group execution-time budgets
- AI-10266 Task termination procedure
- AI-00386 Further functions returning Time_Span values

Ravenscar profile

See also:

- AI-00249 Ravenscar profile for high-integrity systems
- AI-00305 New pragma and additional restriction identifiers for real-time systems
- AI-00347 Title of Annex H
- AI-00265 Partition Elaboration Policy for High-Integrity Systems

New scheduling policies

See also:

- AI-00355 Priority Specific Dispatching including Round Robin
- AI-00357 Support for Deadlines and Earliest Deadline First Scheduling
- AI-00298 Non-Preemptive Dispatching

Dynamic priorities for protected objects

See also: AI-00327 Dynamic ceiling priorities

Summary of what's new

New keywords

- **interface**

- **overriding**
- **synchronized**

New pragmas

- **pragma** Assert
- **pragma** Assertion_Policy
- **pragma** Detect_Blocking
- **pragma** No_Return
- **pragma** Partition_Elaboration_Policy
- **pragma** Preelaborable_Initialization
- **pragma** Priority_Specific_Dispatching
- **pragma** Profile
- **pragma** Relative_Deadline
- **pragma** Unchecked_Union
- **pragma** Unsuppress

New attributes

- Machine_Rounding
- Mod
- Priority
- Stream_Size
- Wide_Wide_Image
- Wide_Wide_Value
- Wide_Wide_Width

New packages

- Assertions:
  - Ada.Assertions

- Container library:
  - Ada.Containers
- Ada.Containers.Vectors
- Ada.Containers.Doubly_Linked_Lists
- Ada.Containers.Generic_Array_Sort (generic procedure)
- Ada.Containers.Generic_Constrained_Array_Sort (generic procedure)
- Ada.Containers.Hashed_Maps
- Ada.Containers.Ordered_Maps
- Ada.Containers.Hashed_Sets
- Ada.Containers.Ordered_Sets
- Ada.Containers.Indefinite_Vectors
- Ada.Containers.Indefinite_Doubly_Linked_Lists
- Ada.Containers.Indefinite_Hashed_Maps
- Ada.Containers.Indefinite_Ordered_Maps
- Ada.Containers.Indefinite_Hashed_Sets
- Ada.Containers.Indefinite_Ordered_Sets

- Vector and matrix manipulation:
  - Ada.Numerics.Real_Arrays
- Ada.Numerics.Complex_Arrays
- Ada.Numerics.Generic_Real_Arrays
- Ada.Numerics.Generic_Complex_Arrays

- General OS facilities:
  - Ada.Directories
- Ada.Directories.Information
- Ada.Environment_Variables

- String hashing:
  - Ada.Strings.Hash (generic function)
- Ada.Strings.Fixed.Hash (generic function)
- Ada.Strings.Bounded.Hash (generic function)
- Ada.Strings.Unbounded.Hash (generic function)
- Ada.Strings.Wide_Hash (generic function)
- Ada.Strings.Wide_Fixed.Wide_Hash (generic function)
- Ada.Strings.Wide_Bounded.Wide_Hash (generic function)
- Ada.Strings.Wide_Unbounded.Wide_Hash (generic function)
- Ada.Strings.Wide_Wide_Hash (generic function)
- Ada.Strings.Wide_Wide_Fixed.Wide_Wide_Hash (generic function)
- Ada.Strings.Wide_Wide_Bounded.Wide_Wide_Hash (generic function)
- Ada.Strings.Wide_Wide_Unbounded.Wide_Wide_Hash (generic function)

- Time operations:
  - Ada.Calendar.Time_Zones
- Ada.Calendar.Arithmetic
- Ada.Calendar.Formatting

- Tagged types:
  - Ada.Tags.Generic_Dispatching_Constructor (generic function)

- Text packages:
  - Ada.Complex_Text_IO
- Ada.Text_IO.Bounded_IO
- Ada.Text_IO.Unbounded_IO
- Ada.Wide_Text_IO.Bounded_IO
- Ada.Wide_Text_IO.Unbounded_IO
- Ada.Wide_Characters
- Ada.Wide_Wide_Characters

- `Wide_Wide_Character` packages:
  - Ada.Strings.Wide_Wide_Bounded
- Ada.Strings.Wide_Wide_Fixed
- Ada.Strings.Wide_Wide_Maps
- Ada.Strings.Wide_Wide_Maps.Wide_Wide_Constants
- Ada.Strings.Wide_Wide_Unbounded
- Ada.Wide_Wide_Text_IO
- Ada.Wide_Wide_Text_IO.Complex_IO
- Ada.Wide_Wide_Text_IO.Editing

- Ada.Wide_Wide_Text_IO.Text_Streams
- Ada.Wide_Wide_Text_IO.Unbounded_IO

- Execution-time clocks:
  - Ada.Execution_Time
- Ada.Execution_Time.Timers
- Ada.Execution_Time.Group_Budgets

- Dispatching:
  - Ada.Dispatching
- Ada.Dispatching.EDF
- Ada.Dispatching.Round_Robin

- Timing events:
  - Ada.Real_Time.Timing_Events

- Task termination procedures:
  - Ada.Task_Termination

See also

Wikibook

- Ada Programming/Object Orientation
- Ada Programming/Types/access
- Ada Programming/Keywords
- Ada Programming/Keywords/and
- Ada Programming/Keywords/interface
- Ada Programming/Attributes
- Ada Programming/Pragmas
- Ada Programming/Pragmas/Restrictions
- Ada Programming/Libraries/Ada.Containers
- Ada Programming/Libraries/Ada.Directories

Pages in the category Ada 2005

- Category:Ada Programming/Ada 2005 feature

External links

Papers and presentations

- Ada 2005: Putting it all together (SIGAda 2004 presentation)
- GNAT and Ada 2005 (SIGAda 2004 paper)
- An invitation to Ada 2005, and the presentation of this paper at Ada-Europe 2004

Rationale

- *Rationale for Ada 2005* by John Barnes:
    1. Introduction
- Object Oriented Model
- Access Types
- Structure and Visibility
- Tasking and Real-Time
- Exceptions, Generics, Etc.
- Predefined Library
- Containers
- Epilogue

References
Index

Available as a single document for printing.

Language Requirements

- *Instructions to the Ada Rapporteur Group from SC22/WG9 for Preparation of the Amendment to ISO/IEC 8652* (10 October 2002), and a presentation of this document at SIGAda 2002

Ada Reference Manual

- **Ada Reference Manual**, ISO/IEC 8652:1995(E) with COR.1:2001 and AMD.1:200y (Draft)
- **Annotated Ada Reference Manual**, ISO/IEC 8652:1995(E) with COR.1:2001 and AMD.1:200y (Draft with colored diffs)
- List of Ada Amendment drafts

Ada Issues

- Amendment 200Y
    - AI-00387 Introduction to Amendment
- AI-10284 New reserved words
- AI-00252 Object.Operation notation
- AI-20218 Accidental overloading when overriding
- AI-00348 Null procedures
- AI-00287 Limited aggregates allowed
- AI-00326 Incomplete types
- AI-00317 Partial parameter lists for formal packages
- AI-00376 Interfaces.C works for C++ as well
- AI-00368 Restrictions for obsolescent features
- AI-00381 New Restrictions identifier No_Dependence
- AI-00224 pragma Unsuppress
- AI-00161 Default-initialized objects

- AI-00361 Raise with message
- AI-00286 Assert pragma
- AI-00328 Preinstantiations of Complex_IO
- AI-00301 Operations on language-defined string types
- AI-00340 Mod attribute
- AI-00364 Fixed-point multiply/divide
- AI-00267 Fast float-to-integer conversions
- AI-00321 Definition of dispatching policies
- AI-00329 pragma No_Return -- procedures that never return
- AI-00362 Some predefined packages should be recategorized
- AI-00351 Time operations
- AI-00427 Default parameters and Calendar operations
- AI-00270 Stream item size control

24 Tips

Full declaration of a type can be deferred to the unit's body

Often, you'll want to make changes to the internals of a private type. This, in turn, will require the algorithms that act on it to be modified. If the type is completed in the unit specification, it is a pain to edit and recompile both files, even with an IDE, but it's something some programmers learn to live with.

It turns out you don't have to. Nonchalantly mentioned in the ARM, and generally skipped over in tutorials, is the fact that private types can be completed in the unit's body itself, making them much closer to the relevant code, and saving a recompile of the specification, as well as every unit depending on it. This may seem like a small thing, and, for small projects, it is. However, if you have one of those uncooperative types that requires dozens of tweaks, or if your dependence graph has much depth, the time and annoyance saved add up quickly.

Also, this construction is very useful when coding a shared library, because it permits to change the implementation of the type while still providing a compatible ABI.

Code sample:

```
package Private_And_Body is

  type Private_Type is limited private;
  -- Operations...

private
  type Body_Type;    -- Defined in the body
  type Private_Type is access Body_Type;
end Private_And_Body;
```

The type in the public part is an access to the hidden type. This has the drawback that memory management has to be provided by the package implementation. That is the reason why `Private_Type` is a limited type, the client will not be allowed to copy the access values, in order to prevent dangling references.

These types are sometimes called "Taft types" —named after Tucker Taft, the main designer of Ada 95— because were introduced in the so-called Taft Amendment to Ada 83. In other programming languages, this technique is called "opaque pointers".

Lambda calculus through generics

Suppose you've decided to roll your own set type. You can add things to it, remove things from it, and you want to let a user apply some arbitrary function to all of its members. But the scoping rules seem to conspire against you, forcing nearly everything to be global.

The mental stumbling block is that most examples given of generics are packages, and the Set package is already generic. In this case, the solution is to make the Apply_To_All procedure generic as well; that is, to nest the generics. Generic procedures inside packages exist in a strange scoping limbo, where anything in scope at the instantiation can be used by the instantiation, and anything normally in scope at the formal can be accessed by the formal. The end result is that the relevant scoping roadblocks no longer apply. It isn't the full lambda calculus, just one of the most useful parts.

```
generic
  type Element is private;
package Sets is
  type Set is private;
   [..]
  generic
    with procedure Apply_To_One (The_Element : in out Element);
  procedure Apply_To_All (The_Set : in out Set);
end Sets;
```

Compiler Messages

Different compilers can diagnose different things differently, or the same thing using different messages, etc.. Having two compilers at hand canbe useful.

`selected component`

When a source program contains a construct such as `Foo.Bar`, you may see messages saying something like «selected component "Bar"» or maybe like «selected component "Foo"». The phrases may seem confusing, because one refers to `Foo`, while the other refers to `Bar`. But they are both right. The reason is that `selected_component` is an item from Ada's grammar (4.1.3 Selected Components (Annotated)). It denotes all of: a prefix, a dot, and a selector_name. In the `Foo.Bar` example these correspond to `Foo`, `'.'`, and `Bar`. Look for more grammar words in the compiler messages, e.g. «prefix», and associate them with identifiers quoted in the messages.

I/O

Text_IO Issues

A canonical method of reading a sequence of lines from a text file uses the standard procedure Ada.Text_IO.*Get_Line*. When the end of input is reached, *Get_Line* will fail, and exception *End_Error* is raised. Some programs will use another function from Ada.Text_IO to prevent this and test for *End_of_Input*. However, this isn't always the best choice, as has been explained for example in a news group discussion on comp.lang.ada (see message `1165428385.365155.179960@j44g2000cwa.googlegroups.com`, 2006-12-06).

A working solution uses an exception handler instead:

```
declare
   The_Line: String(1..100);
   Last: Natural;
begin
   loop
      Text_IO.Get_Line(The_Line, Last);
      --  do something with The_Line ...
   end loop;
exception
   when Text_IO.End_Error =>
      null;
end;
```

See also

Wikibook

- Ada Programming

Ada Reference Manual

- 3.10.1 Incomplete Type Declarations (Annotated)

25 ALGORITHMS

Introduction

Welcome to the Ada implementations of the Algorithms Wikibook. For those who are new to Ada Programming a few notes:

- All examples are fully functional with all the needed input and output operations. However, only the code needed to outline the algorithms at hand is copied into the text - the full samples are available via the download links. (Note: It can take up to 48 hours until the cvs is updated).
- We seldom use predefined types in the sample code but define special types suitable for the algorithms at hand.
- Ada allows for default function parameters; however, we always fill in and name all parameters, so the reader can see which options are available.
- We seldom use shortcuts - like using the attributes Image or Value for String <=> Integer conversions.

All these rules make the code more elaborate then perhaps needed. However, we also hope it makes the code easier to understand

Chapter 1: Introduction

The following subprograms are implementations of the *Inventing an Algorithm* examples.

To Lower

The Ada example code does not append to the array as the algorithms. Instead we create an empty array of the desired length and then replace the characters inside.

File: to_lower_1.adb (view, plain text, download page, browse all)

```
function To_Lower (C : Character) return Character renames
   Ada.Characters.Handling.To_Lower;

--  tolower - translates all alphabetic, uppercase characters
--  in str to lowercase
function To_Lower (Str : String) return String is
   Result : String (Str'Range);
begin
   for C in  Str'Range loop
      Result (C) := To_Lower (Str (C));
   end loop;
   return Result;
end To_Lower;
```

Would the append approach be impossible with Ada? No, but it would be significantly more complex and slower.

Equal Ignore Case

File: to_lower_2.adb (view, plain text, download page, browse all)

```
--  equal-ignore-case -- returns true if s or t are equal,
--  ignoring case
function Equal_Ignore_Case
  (S    : String;
   T    : String)
   return Boolean
is
   O : constant Integer := S'First - T'First;
begin
   if T'Length /= S'Length then
      return False;  --  if they aren't the same length, they
                     --  aren't equal
   else
      for I in  S'Range loop
         if To_Lower (S (I)) /=
            To_Lower (T (I + O))
         then
            return False;
         end if;
      end loop;
   end if;
   return True;
end Equal_Ignore_Case;
```

Chapter 6: Dynamic Programming

Fibonacci numbers

The following codes are implementations of the Fibonacci-Numbers examples.

Simple Implementation

File: fibonacci_1.adb (view, plain text, download page, browse all)

...

To calculate Fibonacci numbers negative values are not needed so we define an integer type which starts at 0. With the integer type defined you can calculate up until `Fib (87)`. `Fib (88)` will result in an `Constraint_Error`.

```
type Integer_Type is range 0 .. 999_999_999_999_999_999;
```

You might notice that there is not equivalence for the `assert (n >= 0)` from the original example. Ada will test the correctness of the parameter *before* the function is called.

```
function Fib (n : Integer_Type) return Integer_Type is
begin
   if n = 0 then
      return 0;
   elsif n = 1 then
```

```
         return 1;
      else
         return Fib (n - 1) + Fib (n - 2);
      end if;
   end Fib;
...
```

Cached Implementation

File: fibonacci_2.adb (view, plain text, download page, browse all)

```
...
```

For this implementation we need a special cache type can also store a -1 as "not calculated" marker

```
type Cache_Type is range -1 .. 999_999_999_999_999_999;
```

The actual type for calculating the fibonacci numbers continues to start at 0. As it is a **subtype** of the cache type Ada will automatically convert between the two. (the conversion is - of course - checked for validity)

```
subtype Integer_Type is Cache_Type range
   0 .. Cache_Type'Last;
```

In order to know how large the cache need to be we first read the actual value from the command line.

```
Value : constant Integer_Type :=
   Integer_Type'Value (Ada.Command_Line.Argument (1));
```

The Cache array starts with element 2 since Fib (0) and Fib (1) are constants and ends with the value we want to calculate.

```
type Cache_Array is
   array (Integer_Type range 2 .. Value) of Cache_Type;
```

The Cache is initialized to the first valid value of the cache type — this is -1.

```
F : Cache_Array := (others => Cache_Type'First);
```

What follows is the actual algorithm.

```
function Fib (N : Integer_Type) return Integer_Type is
begin
   if N = 0 or else N = 1 then
      return N;
   elsif F (N) /= Cache_Type'First then
      return F (N);
   else
      F (N) := Fib (N - 1) + Fib (N - 2);
      return F (N);
   end if;
end Fib;
...
```

This implementation is faithful to the original from the Algorithms book. However, in Ada you would normally do it a little different:

File: fibonacci_3.adb (view, plain text, download page, browse all)

when you use a slightly larger array which also stores the elements 0 and 1 and initializes them to the correct values

```
type Cache_Array is
   array (Integer_Type range 0 .. Value) of Cache_Type;
F : Cache_Array :=
    (0      => 0,
     1      => 1,
     others => Cache_Type'First);
```

and then you can remove the first **if** path.

```
~~~~~~if N = 0 or else N = 1 then
~~~~~~~~return N;~~
~~~~~~elsif F (N) /= Cache_Type'First then
```

This will save about 45% of the execution-time (measured on Linux i686) while needing only two more elements in the cache array.

Memory Optimized Implementation

This version looks just like the original in WikiCode.

File: fibonacci_4.adb (view, plain text, download page, browse all)

```
type Integer_Type is range 0 .. 999_999_999_999_999_999;
function Fib (N : Integer_Type) return Integer_Type is
   U : Integer_Type := 0;
   V : Integer_Type := 1;
begin
   for I in  2 .. N loop
      Calculate_Next : declare
         T : constant Integer_Type := U + V;
      begin
         U := V;
         V := T;
      end Calculate_Next;
   end loop;
   return V;
end Fib;
```

No 64 bit integers

Your Ada compiler does not support 64 bit integer numbers? Then you could try to use decimal numbers instead. Using decimal numbers results in a slower program (takes about three times as long) but the result will be the same.

The following example shows you how to define a suitable decimal type. Do experiment with the **digits** and **range** parameters until you get the optimum out of your Ada compiler.

File: fibonacci_5.adb (view, plain text, download page, browse all)

```
type Integer_Type is delta 1.0 digits 18 range
   0.0 .. 999_999_999_999_999_999.0;
```

You should know that floating point numbers are unsuitable for the calculation of fibonacci numbers. They will not report an error condition when the number calculated becomes too large — instead they will lose in precision which makes the result meaningless.

26 FUNCTION OVERLOADING

Description

Function overloading (also *polymorphism* or *method overloading*) is a programming concept that allows programmers to define two or more functions with the same name.

Each function has a unique signature (or header), which is derived from:

1. function/procedure name
2. number of arguments
3. arguments' type
4. arguments' order
5. arguments' name
6. return type

Please note: Not all above signature options are available in all programming languages.

Available functions overloadings

Language	1	2	3	4	5	6
Ada	yes	yes	yes	yes	yes	yes
C++	yes	yes	yes	yes	no	no
Java	yes	yes	yes	yes	no	no

Warning: Function overloading is often confused with function overriding. In Function overloading, a function with a different signature is created, adding to the pool of available functions. In function overriding, however, a function with the same signature is declared, replacing the old function in the context of the new function.

Demonstration

Since functions' names are in this case the same, we must preserve uniqueness of signatures, by changing something from the ***parameter list*** (last three alienees).

If the functions' signatures are sufficiently different, the compiler can distinguish which function was intended to be used at each occurence. This process of searching for the appropriate function is called ***function resolution*** and can be quite an intensive one, especially if there are a lot of equally named functions.

Programming languages supporting implicid type convertions usually use promotion of arguments (ie.: type casting of integer to floating-point) when there is no exact function match. The demotion of arguments is rarely used.

When two or more functions match the criteria in function resolution process, an ***ambiguity error*** is reported by compiler. Adding more information to compiler by editing the source code (using for example type casting), can address such doubts.

The example code shows how function overloading can be used. As functions do practically the same thing, it makes sense to use function overloading.

File: function_overloading.adb (view, plain text, download page, browse all)

```
function Generate_Number (MaxValue : Integer) return Integer is
   subtype Random_Type is Integer range 0 .. MaxValue;
   package Random_Pack is new Ada.Numerics.Discrete_Random (Random_Type);

   G : Random_Pack.Generator;
begin
   Random_Pack.Reset (G);
   return Random_Pack.Random (G);
end Generate_Number;
function Generate_Number (MinValue : Integer;
                          MaxValue : Integer) return Integer
is
   subtype Random_Type is Integer range MinValue .. MaxValue;
   package Random_Pack is new Ada.Numerics.Discrete_Random (Random_Type);

   G : Random_Pack.Generator;
begin
   Random_Pack.Reset (G);
   return Random_Pack.Random (G);
end Generate_Number;
```

calling the first function

The first code block will generate numbers from 0 up to specified parameter *MaxValue*. The appropriate function call is:

```
Number_1 : Integer := Generate_Number (10);
```

calling the second function

The second requires another parameter *MinValue*. Function will return numbers above or equals *MinValue* and lower than *MaxValue*.

```
Number_2 : Integer := Generate_Number (6, 10);
```

Function overloading in Ada

Ada supports all six signature options but if you use the arguments' name as option you will always have to name the parameter when calling the function. i.e.:

```
Number_2 : Integer := Generate_Number (MinValue => 6,
                                       MaxValue => 10);
```

Note that you cannot overload a generic procedure or generic function within the same package. The following example will fail to compile:

```
package myPackage
  generic
    type Value_Type is (<>);
  -- The first declaration of a generic subprogram
  -- with the name "Generic_Subprogram"
  procedure Generic_Subprogram (Value : in out Value_Type);
  ...
  generic
    type Value_Type is (<>);
  -- This subprogram has the same name, but no
  -- input or output parameters. A non-generic
  -- procedure would be overloaded here.
  -- Since this procedure is generic, overloading
  -- is not allowed and this package will not compile.
  procedure Generic_Subprogram;
  ...
  generic
    type Value_Type is (<>);
  -- The same situation.
  -- Even though this is a function and not
  -- a procedure, generic overloading of
  -- the name "Generic_Subprogram" is not allowed.
  function Generic_Subprogram (Value : Value_Type) return Value_Type;
end myPackage;
```

See also

Wikibook

- Ada Programming
- Ada Programming/Subprograms

Ada 95 Reference Manual

- 6.6 Overloading of Operators (Annotated)
- 8.6 The Context of Overload Resolution (Annotated)

Ada 2005 Reference Manual

- 6.6 Overloading of Operators (Annotated)
- 8.6 The Context of Overload Resolution (Annotated)

27 Mathematical calculations

Ada is very well suited for all kind of calculations. You can define you own fixed point and floating point types and — with the aid of generic packages call all the mathematical functions you need. In that respect Ada is on par with Fortran. This module will show you how to use them and while we progress we create a simple RPN calculator.

Simple calculations

Addition

Additions can be done using the predefined operator +. The operator is predefined for all numeric types and the following, working code, demonstrates its use:

File: numeric_1.adb (view, plain text, download page, browse all)

```
-- The Package Text_IO
with Ada.Text_IO;
procedure Numeric_1 is
   type Value_Type is digits 12
         range -999_999_999_999.0e999 .. 999_999_999_999.0e999;
   package T_IO renames Ada.Text_IO;
   package F_IO is new  Ada.Text_IO.Float_IO (Value_Type);
   Value_1 : Value_Type;
   Value_2 : Value_Type;
begin
   T_IO.Put ("First Value : ");
   F_IO.Get (Value_1);
   T_IO.Put ("Second Value : ");
   F_IO.Get (Value_2);

   F_IO.Put (Value_1);
   T_IO.Put (" + ");
   F_IO.Put (Value_2);
   T_IO.Put (" = ");
   F_IO.Put (Value_1 + Value_2);
end Numeric_1;
```

Subtraction

Subtractions can be done using the predefined operator -. The following extended demo shows the use of + and - operator together:

File: numeric_2.adb (view, plain text, download page, browse all)

```
-- The Package Text_IO
with Ada.Text_IO;

procedure Numeric_2
is
  type Value_Type
  is digits
     12
  range
     -999_999_999_999.0e999 .. 999_999_999_999.0e999;

  package T_IO renames Ada.Text_IO;
  package F_IO is new  Ada.Text_IO.Float_IO (Value_Type);
  Value_1   : Value_Type;
  Value_2   : Value_Type;
  Result    : Value_Type;
```

```
   Operation : Character;
begin
   T_IO.Put ("First Value  : ");
   F_IO.Get (Value_1);
   T_IO.Put ("Second Value : ");
   F_IO.Get (Value_2);
   T_IO.Put ("Operation    : ");
   T_IO.Get (Operation);

   case Operation is
      when '+' =>
         Result := Value_1 + Value_2;
      when '-' =>
         Result := Value_1 - Value_2;
      when others =>
         T_IO.Put_Line ("Illegal Operation.");
         goto Exit_Numeric_2;
   end case;
   F_IO.Put (Value_1);
   T_IO.Put (" ");
   T_IO.Put (Operation);
   T_IO.Put (" ");
   F_IO.Put (Value_2);
   T_IO.Put (" = ");
   F_IO.Put (Result);
<<Exit_Numeric_2>>
   return;
end Numeric_2;
```

Purists might be surprised about the use of goto - but some people prefer the use of goto over the use of multiple return statements if inside functions - given that, the opinions on this topic vary strongly. See the isn't goto evil article.

Multiplication

Multiplication can be done using the predefined operator *. For a demo see the next chapter about Division.

Division

Divisions can be done using the predefined operators /, mod, rem. The operator / performs a normal division, mod returns a modulus division and rem returns the remainder of the modulus division.

The following extended demo shows the use of the +, -, * and / operators together as well as the use of an four number wide stack to store intermediate results:

The operators mod and rem are not part of the demonstration as they are only defined for integer types.

File: numeric_3.adb (view, plain text, download page, browse all)

```
with Ada.Text_IO;
procedure Numeric_3 is
   procedure Pop_Value;
   procedure Push_Value;
   type Value_Type is digits 12 range
     -999_999_999_999.0e999 .. 999_999_999_999.0e999;

   type Value_Array is array (Natural range 1 .. 4) of Value_Type;
   package T_IO renames Ada.Text_IO;
   package F_IO is new Ada.Text_IO.Float_IO (Value_Type);
   Values    : Value_Array := (others => 0.0);
   Operation : String (1 .. 40);
   Last      : Natural;
   procedure Pop_Value is
```

```
      begin
         Values (Values'First + 1 .. Values'Last) :=
           Values (Values'First + 2 .. Values'Last) & 0.0;
      end Pop_Value;
      procedure Push_Value is
      begin
         Values (Values'First + 1 .. Values'Last) :=
           Values (Values'First .. Values'Last - 1);
      end Push_Value;
   begin
      Main_Loop:
      loop
         T_IO.Put (">");
         T_IO.Get_Line (Operation, Last);

         if Last = 1 and then Operation (1) = '+' then
            Values (1) := Values (1) + Values (2);
            Pop_Value;
         elsif Last = 1 and then Operation (1) = '-' then
            Values (1) := Values (1) + Values (2);
            Pop_Value;
         elsif Last = 1 and then Operation (1) = '*' then
            Values (1) := Values (1) * Values (2);
            Pop_Value;
         elsif Last = 1 and then Operation (1) = '/' then
            Values (1) := Values (1) / Values (2);
            Pop_Value;
         elsif Last = 4 and then Operation (1 .. 4) = "exit" then
            exit Main_Loop;
         else
            Push_Value;
            F_IO.Get (From => Operation, Item => Values (1), Last => Last);
         end if;

         Display_Loop:
         for I in reverse Value_Array'Range loop
            F_IO.Put
              (Item => Values (I),
               Fore => F_IO.Default_Fore,
               Aft  => F_IO.Default_Aft,
               Exp  => 4);
            T_IO.New_Line;
         end loop Display_Loop;
      end loop Main_Loop;
      return;
   end Numeric_3;
```

Exponential calculations

All exponential functions are defined inside the generic package Ada.Numerics.Generic_Elementary_Functions.

Power of

Calculation of the form x^y are performed by the operator **. Beware: There are two versions of this operator. The predefined operator ** allows only for Standard.Integer to be used as exponent. If you need to use a floating point type as exponent you need to use the ** defined in Ada.Numerics.Generic_Elementary_Functions.

Root

The square root $\sqrt{x}$ is calculated with the function `Sqrt()`. There is no function defined to calculate an arbitrary root $\sqrt[n]{x}$. However you can use logarithms to calculate an arbitrary root using the mathematical identity: $\sqrt[b]{a} = e^{\log_e(a)/b}$ which will become `root := Exp (Log (a) / b)` in

Ada. Alternatively, use $\sqrt[b]{a} = a^{\frac{1}{b}}$ which, in Ada, is `root := a**(1.0/b)`.

Logarithm

Ada.Numerics.Generic_Elementary_Functions defines a function for both the arbitrary logarithm $log_n(x)$ and the natural logarithm $log_e(x)$ both of of which have the same name `Log()` distinguished by the amount ofparameters.

Demonstration

The following extended demo shows the how to use the exponential functions in Ada. The new demo also uses Unbounded_String instead of Strings which make the comparisons easier.

Please note that from now on we won't copy the full sources any more. Do follow the download links to see the full program.

File: numeric_4.adb (view, plain text, download page, browse all)

```
with Ada.Text_IO;
with Ada.Numerics.Generic_Elementary_Functions;
with Ada.Strings.Unbounded;
procedure Numeric_4 is
  package Str renames Ada.Strings.Unbounded;
  package T_IO renames Ada.Text_IO;
  procedure Pop_Value;
  procedure Push_Value;
  function Get_Line return Str.Unbounded_String;
  type Value_Type is digits 12 range
     -999_999_999_999.0e999 .. 999_999_999_999.0e999;

  type Value_Array is array (Natural range 1 .. 4) of Value_Type;
  package F_IO is new Ada.Text_IO.Float_IO (Value_Type);
  package Value_Functions is new Ada.Numerics.Generic_Elementary_Functions (
     Value_Type);
  use Value_Functions;
  use type Str.Unbounded_String;
  Values    : Value_Array := (others => 0.0);
  Operation : Str.Unbounded_String;
  Dummy     : Natural;
  function Get_Line return Str.Unbounded_String is
     BufferSize : constant := 2000;
     Retval     : Str.Unbounded_String := Str.Null_Unbounded_String;
     Item       : String (1 .. BufferSize);
     Last       : Natural;
  begin
     Get_Whole_Line :
        loop
           T_IO.Get_Line (Item => Item, Last => Last);

           Str.Append (Source => Retval, New_Item => Item (1 .. Last));

           exit Get_Whole_Line when Last < Item'Last;
        end loop Get_Whole_Line;
     return Retval;
  end Get_Line;
...
begin
  Main_Loop :
     loop
        T_IO.Put (">");
        Operation := Get_Line;

...
        elsif Operation = "e" then
           --  insert e
           Push_Value;
```

```
            Values (1) := Ada.Numerics.e;
         elsif Operation = "**" or else Operation = "^" then
            --  power of x^y
            Values (1) := Values (1) ** Values (2);
            Pop_Value;
         elsif Operation = "sqr" then
            --  square root
            Values (1) := Sqrt (Values (1));
         elsif Operation = "root" then
            --  arbritary root
            Values (1) :=
               Exp (Log (Values (2)) / Values (1));
            Pop_Value;
         elsif Operation = "ln" then
            --  natural logarithm
            Values (1) := Log (Values (1));
         elsif Operation = "log" then
            --  based logarithm
            Values (1) :=
               Log (Base => Values (1), X => Values (2));
            Pop_Value;
         elsif Operation = "exit" then
            exit Main_Loop;
         else
            Push_Value;
            F_IO.Get
              (From => Str.To_String (Operation),
               Item => Values (1),
               Last => Dummy);
         end if;
...
      end loop Main_Loop;
   return;
end Numeric_4;
```

Higher math

Trigonometric calculations

The full set of trigonometric functions are defined inside the generic package Ada.Numerics.Generic_Elementary_Functions. All functions are defined for 2π and an arbitrary cycle value (a full cycle of revolution).

Please note the difference of calling the `Arctan ()` function.

File: numeric_5.adb (view, plain text, download page, browse all)

```
with Ada.Text_IO;
with Ada.Numerics.Generic_Elementary_Functions;
with Ada.Strings.Unbounded;
procedure Numeric_5 is

...

   procedure Put_Line (Value : in Value_Type);
   use Value_Functions;
   use type Str.Unbounded_String;
   Values    : Value_Array := (others => 0.0);
   Cycle     : Value_Type  := Ada.Numerics.Pi;
   Operation : Str.Unbounded_String;
   Dummy     : Natural;
...
   procedure Put_Line (Value : in Value_Type) is
   begin
      if abs Value_Type'Exponent (Value) >=
         abs Value_Type'Exponent (10.0 ** F_IO.Default_Aft)
      then
         F_IO.Put
           (Item => Value,
            Fore => F_IO.Default_Aft,
            Aft  => F_IO.Default_Aft,
```

```
                Exp  => 4);
         else
            F_IO.Put
              (Item => Value,
               Fore => F_IO.Default_Aft,
               Aft  => F_IO.Default_Aft,
               Exp  => 0);
         end if;
         T_IO.New_Line;
         return;
      end Put_Line;
   ...
   begin
      Main_Loop :
         loop
            Display_Loop :
               for I in reverse  Value_Array'Range loop
                  Put_Line (Values (I));
               end loop Display_Loop;
            T_IO.Put (">");
            Operation := Get_Line;
   ...
            elsif Operation = "deg" then
               --  switch to degrees
               Cycle := 360.0;
            elsif Operation = "rad" then
               --  switch to degrees
               Cycle := Ada.Numerics.Pi;
            elsif Operation = "grad" then
               --  switch to degrees
               Cycle := 400.0;
            elsif Operation = "pi" or else Operation = "π" then
               --  switch to degrees
               Push_Value;
               Values (1) := Ada.Numerics.Pi;
            elsif Operation = "sin" then
               --  sinus
               Values (1) := Sin (X => Values (1), Cycle => Cycle);
            elsif Operation = "cos" then
               --  cosinus
               Values (1) := Cos (X => Values (1), Cycle => Cycle);
            elsif Operation = "tan" then
               --  tangents
               Values (1) := Tan (X => Values (1), Cycle => Cycle);
            elsif Operation = "cot" then
               --  cotanents
               Values (1) := Cot (X => Values (1), Cycle => Cycle);
            elsif Operation = "asin" then
               --  arc-sinus
               Values (1) := Arcsin (X => Values (1), Cycle => Cycle);
            elsif Operation = "acos" then
               --  arc-cosinus
               Values (1) := Arccos (X => Values (1), Cycle => Cycle);
            elsif Operation = "atan" then
               --  arc-tangents
               Values (1) := Arctan (Y => Values (1), Cycle => Cycle);
            elsif Operation = "acot" then
               --  arc-cotanents
               Values (1) := Arccot (X => Values (1), Cycle => Cycle);

   ...
         end loop Main_Loop;
      return;
   end Numeric_5;
```

The Demo also contains an improved numeric output which behaves more like a normal calculator.

Hyperbolic calculations

You guessed it: The full set of hyperbolic functions is defined inside the generic package Ada.Numerics.Generic_Elementary_Functions.

File: numeric_6.adb (view, plain text, download page, browse all)

```
with Ada.Text_IO;
with Ada.Numerics.Generic_Elementary_Functions;
with Ada.Strings.Unbounded;
with Ada.Exceptions;
procedure Numeric_6 is
  package Str renames Ada.Strings.Unbounded;
  package T_IO renames Ada.Text_IO;
  package Exept renames Ada.Exceptions;
...
 begin
  Main_Loop :
     loop
        Try :
           begin
              Display_Loop :
...
              elsif Operation = "sinh" then
                 --  sinus hyperbolic
                 Values (1) := Sinh (Values (1));
              elsif Operation = "cosh" then
                 --  cosinus hyperbolic
                 Values (1) := Coth (Values (1));
              elsif Operation = "tanh" then
                 --  tangents hyperbolic
                 Values (1) := Tanh (Values (1));
              elsif Operation = "coth" then
                 --  cotanents hyperbolic
                 Values (1) := Coth (Values (1));
              elsif Operation = "asinh" then
                 --  arc-sinus hyperbolic
                 Values (1) := Arcsinh (Values (1));
              elsif Operation = "acosh" then
                 --  arc-cosinus hyperbolic
                 Values (1) := Arccosh (Values (1));
              elsif Operation = "atanh" then
                 --  arc-tangents hyperbolic
                 Values (1) := Arctanh (Values (1));
              elsif Operation = "acoth" then
                 --  arc-cotanents hyperbolic
                 Values (1) := Arccoth (Values (1));
...
           exception
              when An_Exception : others =>
                 T_IO.Put_Line
                   (Exept.Exception_Information (An_Exception));
           end Try;
     end loop Main_Loop;
  return;
end Numeric_6;
```

As added bonus this version supports error handling and therefore won't just crash when an illegal calculation is performed.

Complex arithmethic

For complex arithmetic Ada provides the package Ada.Numerics.Generic_Complex_Types. This package is part of the "special need Annexes" which means it is optional. The open source Ada compiler GNAT implements all "special need Annexes" and therefore has complex arithmetic available.

Since Ada support user defined operators all (+, -, *) operators have there usual meaning as soon as the package Ada.Numerics.Generic_Complex_Types has been instantiated (**package** ... **is new** ...) and the type has been made visible (**use type** ...)

Ada also provides the packages Ada.Text_IO.Complex_IO and Ada.Numerics.Generic_Complex_Elementary_Functions which provide similar functionality to there normal counterparts. But there are some differences:

- Ada.Numerics.Generic_Complex_Elementary_Functions supports only the exponential and trigonometric functions which make sense in complex arithmetic.

- Ada.Text_IO.Complex_IO is a child package of Ada.Text_IO and therefore needs it's own **with**. Note: the `Ada.Text_IO.Complex_IOGet ()` function is pretty fault tolerant - if you forget the "," or the "()" pair it will still parse the input correctly.

So, with only a very few modifications you can convert your "normals" calculator to a calculator for complex arithmetic:

File: numeric_7.adb (view, plain text, download page, browse all)

```
with Ada.Text_IO.Complex_IO;
with Ada.Numerics.Generic_Complex_Types;
with Ada.Numerics.Generic_Complex_Elementary_Functions;
with Ada.Strings.Unbounded;
with Ada.Exceptions;
procedure Numeric_7 is

...

  package Complex_Types is new Ada.Numerics.Generic_Complex_Types (
     Value_Type);
  package Complex_Functions is new
     Ada.Numerics.Generic_Complex_Elementary_Functions (
     Complex_Types);
  package C_IO is new Ada.Text_IO.Complex_IO (Complex_Types);
  type Value_Array is
     array (Natural range 1 .. 4) of Complex_Types.Complex;
  procedure Put_Line (Value : in Complex_Types.Complex);
  use type Complex_Types.Complex;
  use type Str.Unbounded_String;
  use Complex_Functions;
  Values    : Value_Array :=
     (others => Complex_Types.Complex'(Re => 0.0, Im => 0.0));
...

  procedure Put_Line (Value : in Complex_Types.Complex) is
  begin
     if (abs Value_Type'Exponent (Value.Re) >=
         abs Value_Type'Exponent (10.0 ** C_IO.Default_Aft))
       or else (abs Value_Type'Exponent (Value.Im) >=
                abs Value_Type'Exponent (10.0 ** C_IO.Default_Aft))
     then
        C_IO.Put
          (Item => Value,
           Fore => C_IO.Default_Aft,
           Aft  => C_IO.Default_Aft,
           Exp  => 4);
     else
        C_IO.Put
          (Item => Value,
           Fore => C_IO.Default_Aft,
           Aft  => C_IO.Default_Aft,
           Exp  => 0);
     end if;
     T_IO.New_Line;
     return;
  end Put_Line;
begin

...
               elsif Operation = "e" then
                  --  insert e
                  Push_Value;
                  Values (1) :=
                     Complex_Types.Complex'(Re => Ada.Numerics.e, Im => 0.0);

...

               elsif Operation = "pi" or else Operation = "π" then
                  --  insert pi
                  Push_Value;
                  Values (1) :=
                     Complex_Types.Complex'(Re => Ada.Numerics.Pi, Im => 0.0);
               elsif Operation = "sin" then
                  --  sinus
                  Values (1) := Sin (Values (1));
               elsif Operation = "cos" then
                  --  cosinus
```

```
                  Values (1) := Cot (Values (1));
               elsif Operation = "tan" then
                  --  tangents
                  Values (1) := Tan (Values (1));
               elsif Operation = "cot" then
                  --  cotanents
                  Values (1) := Cot (Values (1));
               elsif Operation = "asin" then
                  --  arc-sinus
                  Values (1) := Arcsin (Values (1));
               elsif Operation = "acos" then
                  --  arc-cosinus
                  Values (1) := Arccos (Values (1));
               elsif Operation = "atan" then
                  --  arc-tangents
                  Values (1) := Arctan (Values (1));
               elsif Operation = "acot" then
                  --  arc-cotanents
                  Values (1) := Arccot (Values (1));

   ...

     return;
   end Numeric_7;
```

Vector and Matrix Arithmetic

Ada supports vector and matrix Arithmetic for both normal real types and complex types. For those, the generic packages Ada.Numerics.Generic_Real_Arrays and Ada.Numerics.Generic_Complex_Arrays are used. Both packages offer the usual set of operations, however there is no I/O package and understandably, no package for elementary functions.

Since there is no I/O package for vector and matrix I/O creating a demo is by far more complex — and hence not ready yet. You can have a look at the current progress which will be a universal calculator merging all feature.

Status: Stalled - for a Vector and Matrix stack we need Indefinite_Vectors — which are currently not part of GNAT/Pro. Well I could use the booch components ...

File: numeric_8-complex_calculator.ada (view, plain text, download page, browse all)
File: numeric_8-get_line.ada (view, plain text, download page, browse all)
File: numeric_8-real_calculator.ada (view, plain text, download page, browse all)
File: numeric_8-real_vector_calculator.ada (view, plain text, download page, browse all)

See also

Wikibook

- Ada Programming
- Ada Programming/Delimiters/-
- Ada Programming/Libraries/Ada.Numerics.Generic_Complex_Types
- Ada Programming/Libraries/Ada.Numerics.Generic_Elementary_Functions

Ada 95 Reference Manual

- 4.4 Expressions (Annotated)
- Annex A.5-1 Elementary Functions (Annotated)
- Annex A.10-1 The Package Text_IO (Annotated)
- Annex G.1 Complex Arithmetic (Annotated)
- Annex G.3 Vector and Matrix Manipulation (Annotated)

Ada 2005 Reference Manual

- 4.4 Expressions (Annotated)
- Annex A.5.1 Elementary Functions (Annotated)
- Annex A.10.1 The Package Text_IO (Annotated)
- Annex G.1 Complex Arithmetic (Annotated)
- Annex G.3 Vector and Matrix Manipulation (Annotated)

28 STATEMENTS

Note: there are some simplifications in the explanations below. Don't take anything too literally.

Most programming languages have the concept of a statement. A ***statement*** is a command that the programmer gives to the computer. For example:

```
Ada.Text_IO.Put_Line ("Hi there!");
```

This command has a verb ("Put_Line") and other details (what to print). In this case, the command `"Put_Line"` means "show on the screen," not "print on the printer." The programmer either gives the statement directly to the computer (by typing it while running a special program), or creates a text file with the command in it. You could create a file called "hi.txt", put the above command in it, and give the file to the computer.

If you have more than one command in the file, each will be performed in order, top to bottom. So the file could contain:

```
Ada.Text_IO.Put_Line ("Hi there!");
Ada.Text_IO.Put_Line ("Strange things are afoot...");
```

This does seem like a lot of typing but don't worry: Ada allows you to declare shorter aliasnames if you need a long statement very often.

The computer will perform each of these commands sequentially. It's invaluable to be able to "play computer" when programming. Ask yourself, "If I were the computer, what would I do with these statements?" If you're not sure what the answer is, then you are very likely to write incorrect code. Stop and check the manual for the programming language you're using.

In the above case, the computer will look at the first statement, determine that it's a `Put_Line` statement, look at what needs to be printed, and display that text on the computer screen. It'll look like this:

```
Hi there!
```

Note that the quotation marks aren't there. Their purpose in the program is to tell the computer where the text begins and ends, just like in English prose. The computer will then continue to the next statement, perform its command, and the screen will look like this:

```
Hi there!
Strange things are afoot...
```

When the computer gets to the end of the text file, it stops. There are many different kinds of statements, depending on which programming language is being used. For example, there could be a `beep` statement that causes the computer to output a beep on its speaker, or a `window` statement that causes a new window to pop up.

Also, the way statements are written will vary depending on the programming language. These differences are fairly superficial. The set of rules like the first two is called a programming language's ***syntax***. The set of verbs is called its ***library***.

This article is available in pseudocode, Ada, C, C++, Delphi and Python - do have a look at the other

languages as well.

29 VARIABLES

Variables are *references* that stand in for a *value* that is contained at a certain memory address.

Variables are said to have a value and *may* have a data type. If a variable has a type, then only values of this type may be assigned to it. Variables do not always have atype.

A value can have many values of many different types: integers (7), ratios (1/2), (approximations of) reals (10.234), complex numbers (4+2i), characters ('a'), strings ("hello"), and much more.

Different languages use different names for their types and may not include any of the above.

Assignment statements

An *assignment statement* is used to set a variable to a new value.

Assignment statements are written as *name* **:=** *value*.

```
X := 10;
```

The example set the variable *X* to the integer value of *10*. The assignment statement overwrites the contents of the variable and theprevious value is lost.

In some languages, before a variable can be used, it will have to be declared, where the declaration specifies the type.

Ada is the same. The declaration is as follows:

```
declare
  X : Integer := 10;
begin
  Do_Something (X);
end;
```

Uses

Variables store everything in your program. The purpose of any useful program is to modify variables.

See also

Ada Reference Manual

- 3.3 Objects and Named Numbers (Annotated)

30 Lexical elements

Character set

The character set used in Ada programs is composed of:

- Upper-case letters: A, ..., Z and lower-case letters: a, ..., z.
- Digits: 0, ..., 9.
- Special characters.

Take into account that in Ada 95 the letter range includes accented characters and other letters used in Western Europe languages, those belonging to the *ISO Latin-1* character set, as ç, ñ, ð, etc.

In Ada 2005 the character set has been extended to the full Unicode set, so the identifiers and comments can be written in almost any language in the world.

Ada is a case-insensitive language, i. e. the upper-case set is equivalent to the lower-case set except in character string literals and character literals.

Lexical elements

In Ada we can find the following lexical elements:

- Identifiers
- Numeric Literals
- Character Literals
- String Literals
- Delimiters
- Comments
- Reserved Words

Example:

```
Temperature_In_Room := 25;  -- Temperature to be preserved in the room.
```

This line contains 5 lexical elements:

- The identifier `Temperature_In_Room`.
- The compound delimiter `:=`.
- The number `25`.
- The single delimiter `;`.
- The comment `-- Temperature to be preserved in the room.`.

Identifiers

Definition in *BNF*:

```
identifier ::= letter { [ underscore ] letter | digit }
letter ::= A | ... | Z | a | ... | z
digit ::= 0 | ... | 9
underscore ::= _
```

From this definition we must exclude the keywords that are reserved words in the language and cannot be used as identifiers.

Examples:

The following words are legal Ada identifiers:

```
Time_Of_Day  TimeOfDay  El_Niño_Forecast  Façade  counter ALARM
```

The following ones are **NOT** legal Ada identifiers:

```
_Time_Of_Day  2nd_turn  Start_  Access  Price_In_$  General__Alarm
```

Exercise: could you give the reason for not being legal for each one of them?

Numbers

The numeric literals are composed of the following characters:

- digits `0 .. 9`
- the decimal separator `.`,
- the exponentiation sign `e` or `E`,
- the negative sign `-` and
- the underscore `_`.

The underscore is used as separator for improving legibility for humans, but it is ignored by the compiler. You can separate numbers following any rationale, e.g. decimal integers in groups of three digits, or binary integers in groups of eight digits.

For example, the real number such as 98.4 can be represented as: `9.84E1`, `98.4e0`, `984.0e-1` or `0.984E+2`, but not as `984e-1`.

For integer numbers, for example 1900, it could be written as `1900`, `19E2`, `190e+1` or `1_900E+0`.

A numeric literal could also be expressed in a base different to 10, by enclosing the number between `#` characters, and preceding it by the base, which can be a number between 2 and 16. For example, `2#101#` is 101_2, that is 5_{10}. Another example, with exponent would be `16#B#E2`, that is $11 \times 16^2 = 2{,}816$.

Character literals

Their type is Standard.Character. They are delimited by an apostrophe (').

Examples:

```
'A' 'n' '%'
```

String literals

String literals are of type Standard.String. They are delimited by the quotation mark (").

Example:

```
"This is a string literal"
```

Delimiters

Single delimiters are one of the following special characters:

```
&    '    (    )    *    +    ,    -    .    /    :    ;    <    =    >
```

Compound delimiters are composed of two special characters, and they are the following ones:

```
=>    ..    **    :=    /=    >=    <=    <<    >>    <>
```

You can see a full reference of the delimiters in Ada Programming/Delimiters.

Comments

Comments in Ada start with two consecutive hyphens (- -) and end in the end of line.

```
-- This is a comment in a full line
My_Savings := My_Savings * 10.0; -- This is a comment in a line after a sentece
My_Savings := My_Savings * -- This is a comment inserted inside a sentence
    1_000_000.0;
```

A comment can appear where an end of line can be inserted.

Reserved words

Reserved words are equivalent in upper-case and lower-case letters, although the typical style is the one from the Reference Manual, that is to write them in all lower-case letters.

In Ada some keywords have a different meaning depending on context. You can refer to Ada Programming/Keywords and the following pages for each keyword.

Ada Keywords

abort	else	new	return
abs	elsif	not	reverse
abstract	end	null	
accept	entry		select
access	exception	of	separate
aliased	exit	or	subtype
all		others	synchronized
and	for	out	
array	function	overriding	tagged
at			task
	generic	package	terminate
begin	goto	pragma	then
body		private	type
	if	procedure	
case	in	protected	until
constant	interface		use
	is	raise	
declare		range	when
delay	limited	record	while
delta	loop	rem	with
digits		renames	
do	mod	requeue	xor

See also

Wikibook

- Ada Programming
- Ada Programming/Delimiters
- Ada Programming/Keywords

Ada Reference Manual

- Section 2: Lexical Elements (Annotated)
- 2.1 Character Set (Annotated)
- 2.2 Lexical Elements, Separators, and Delimiters (Annotated)

31 KEYWORDS

Language summary keywords

Most Ada **keywords** have different functions depending on where they are used. A good example is **for** which controls the representation clause when used within a declaration part and controls a loop when used within an implementation.

In Ada, a keyword is also a reserved word, so it cannot be used as an identifier.

List of keywords

Ada Keywords			
abort	else	new	return
abs	elsif	not	reverse
abstract	end	null	
accept	entry		select
access	exception	of	separate
aliased	exit	or	subtype
all		others	synchronized
and	for	out	
array	function	overriding	tagged
at			task
	generic	package	terminate
begin	goto	pragma	then
body		private	type
	if	procedure	
case	in	protected	until
constant	interface		use
	is	raise	
declare		range	when
delay	limited	record	while
delta	loop	rem	with
digits		renames	
do	mod	requeue	xor

See also

Wikibook

- Ada Programming

Ada 95 Reference Manual

- 2.9 Reserved Words (Annotated)
- Annex P (informative) Syntax Summary (Annotated)

Ada 2005 Reference Manual

- 2.9 Reserved Words (Annotated)
- Annex P (informative) Syntax Summary (Annotated)

Ada Quality and Style Guide

- 3.1.3 Capitalization

32 Delimiters

Single character delimiters

&
: ampersand (also operator &)

'
: apostrophe, tick

(
: left parenthesis

)
: right parenthesis

*
: asterisk, multiply (also operator *)

+
: plus sign (also operator +)

,
: comma

-
: hyphen, minus (also operator -)

.
: full stop, point, dot

/
: solidus, divide (also operator /)

:
: colon

;
: semicolon

<
: less than sign (also operator)

=
: equal sign (also operator =)

>
: greater than sign (also operator)

|
: vertical line

Compound character delimiters

=>
: arrow

..
: double dot

**
: double star, exponentiate (also operator **)

:=
: assignment

/=

/=
: inequality (also operator)

>=
: greater than or equal to (also operator)

<=
: less than or equal to (also operator)

<<
: left label bracket

>>
: right label bracket

<>
: box

Others

The following ones are special characters but not delimiters.

"
: quotation mark, used in string literals.

#
: number sign, used in number literals with base.

The following special characters are reserved but currently unused in Ada:

[
: left square bracket

]
: right square bracket

{
: left curly bracket

}
: right curly bracket

See also

Wikibook

- Ada Programming

Ada 95 Reference Manual

- 2.1 Character Set (Annotated)
- 2.2 Lexical Elements, Separators, and Delimiters (Annotated)

Ada 2005 Reference Manual

- 2.1 Character Set (Annotated)
- 2.2 Lexical Elements, Separators, and Delimiters (Annotated)

33 Operators

Standard operators

Ada allows operator overloading for all standard operators and so the following summaries can only describe the suggested standard operations for each operator. It is quite possible to misuse any standard operator to perform something unusual.

Each operator is either a keyword or a delimiter -- hence all operator pages are redirects to the appropiate keyword or delimiter.

The list is sorted from lowest precedence to highest precedence.

Logical operators

and

and $x \wedge y$, (also keyword and)

or

or $x \vee y$, (also keyword or)

xor

exclusive or $(x \wedge \bar{y}) \vee (\bar{x} \wedge y)$, (also keyword xor)

Relational operators

/=

Not Equal $x \neq y$, (also special character /=)

=

Equal $x = y$, (also special character =)

<

Less than $x < y$, (also special character <)

<=

Less than or equal to ($x \leq y$), (also special character <=)

>

Greater than ($x > y$), (also special character >)

>=

Greater than or equal to ($x \geq y$), (also special character >=)

Binary adding operators

+

+
: Add $x + y$, (also special character +)

-
: Subtract $x - y$, (also special character -)

&
: Concatenate , x &  , (also special character &)

Unary adding operators

+
: Plus sign $+x$, (also special character +)

-
: Minus sign $-x$, (also special character -)

Multiplying operator

*
: Multiply, $x \times y$, (also special character *)

/
: Divide x/y, (also special character /)

mod
: modulus (also keyword mod)

rem
: remainder (also keyword rem)

Highest precedence operator

**
: Power x^y, (also special character **)

not
: logical not $\neg x$, (also keyword not)

abs
: absolute value $|x|$ (also keyword abs)

Shortcut operators

The shortcut operators cannot be overloaded.

and then

e.g. `if Y /= 0 and then X/Y > Limit then ...`

or else

e.g. `if Ptr = null or else Ptr.I = 0 then ...`

Membership tests

The Membership Tests also cannot be overloaded.

in

element of, $var \in type$, *e.g.* `if I in Positive then`, (also keyword **in**)

not in

not element of, $var \notin type$, *e.g.* `if I not in Positive then`, (also keywords **not in**)

See also

Wikibook

- Ada Programming

Ada 95 Reference Manual

- 4.5 Operators and Expression Evaluation (Annotated)

Ada 2005 Reference Manual

- 4.5 Operators and Expression Evaluation (Annotated)

Ada Quality and Style Guide

- 2.1.3 Alignment of Operators
- 5.7.4 Overloaded Operators
- 5.7.5 Overloading the Equality Operator

Ada Operators					
`and`	`and then`	`>`	`+`	`abs`	`&`
`or`	`or else`	`>=`	`-`	`mod`	
`xor`	`=`	`<`	`*`	`rem`	`in`
`not`	`/=`	`<=`	`**`	`/`	`not in`

34 Attributes

Language summary attributes

The concept of **attributes** is pretty unique to Ada. Attributes allow you to get —and sometimes set — information about objects or other language entities such as types. A good example is the Size attribute. It describes the size ofan object or a type in bits.

```
A : Natural := Integer'Size; -- A is now 32 (with the GNAT compiler for the x86 architecture)
```

However, unlike the **sizeof** operator from C/C++ the Size attribute can also be set:

```
type Byte is range -128 .. 127;  -- The range fits into 8 bits but the
                                 -- compiler is still free to choose.
for  Byte'Size use 8;            -- Now we force the compiler to use 8 bits.
```

Of course not all attributes can be set. An attribute starts with a tick ' and is followed by its name. The compiler determines by context if the tick isthe beginning of an attribute or of a character literal.

```
A : Character := Character'Val (32) -- A is now a space
B : Character := ' ';               -- B is also a space
```

List of language defined attributes

Ada 2005
: This is a new Ada 2005 attribute.

Obsolescent
: This is a deprecated attribute and should not be used in new code.

A – B

- 'Access
- 'Address
- 'Adjacent
- 'Aft
- 'Alignment
- 'Base
- 'Bit_Order
- 'Body_Version

C

- 'Callable
- 'Caller
- 'Ceiling
- 'Class
- 'Component_Size
- 'Compose

- 'Constrained
- 'Copy_Sign
- 'Count

D – F

- 'Definite
- 'Delta
- 'Denorm
- 'Digits
- 'Emax (Obsolescent)
- 'Exponent
- 'External_Tag
- 'Epsilon (Obsolescent)
- 'First
- 'First_Bit
- 'Floor
- 'Fore
- 'Fraction

G – L

- 'Identity
- 'Image
- 'Input
- 'Large (Obsolescent)
- 'Last
- 'Last_Bit
- 'Leading_Part
- 'Length

M

- 'Machine
- 'Machine_Emax
- 'Machine_Emin
- 'Machine_Mantissa
- 'Machine_Overflows
- 'Machine_Radix
- 'Machine_Rounding (Ada 2005)
- 'Machine_Rounds
- 'Mantissa (Obsolescent)
- 'Max
- 'Max_Size_In_Storage_Elements
- 'Min
- 'Mod (Ada 2005)
- 'Model
- 'Model_Emin

- 'Model_Epsilon
- 'Model_Mantissa
- 'Model_Small
- 'Modulus

O – R

- 'Output
- 'Partition_ID
- 'Pos
- 'Position
- 'Pred
- 'Priority (Ada 2005)
- 'Range
- 'Read
- 'Remainder
- 'Round
- 'Rounding

S

- 'Safe_Emax (Obsolescent)
- 'Safe_First
- 'Safe_Large (Obsolescent)
- 'Safe_Last
- 'Safe_Small (Obsolescent)
- 'Scale
- 'Scaling
- 'Signed_Zeros
- 'Size
- 'Small
- 'Storage_Pool
- 'Storage_Size
- 'Stream_Size (Ada 2005)
- 'Succ

T – V

- 'Tag
- 'Terminated
- 'Truncation
- 'Unbiased_Rounding
- 'Unchecked_Access
- 'Val
- 'Valid
- 'Value
- 'Version

W – Z

- 'Wide_Image
- 'Wide_Value
- 'Wide_Wide_Image (Ada 2005)
- 'Wide_Wide_Value (Ada 2005)
- 'Wide_Wide_Width (Ada 2005)
- 'Wide_Width
- 'Width
- 'Write

List of implementation defined attributes

The following attributes are not available in all Ada compilers, only in those that had implemented them.

Currently, there are only listed the implementation defined attributes of the GNAT compiler. You can help Wikibooks adding implementation dependent attributes of other compilers:

GNAT
: Implementation defined attribute of the GNAT compiler.

DEC Ada
: Implementation defined attribute of the DEC Ada compiler.

A – D

- 'Abort_Signal (GNAT)
- 'Address_Size (GNAT)
- 'Asm_Input (GNAT)
- 'Asm_Output (GNAT)
- 'AST_Entry (GNAT, DEC Ada)
- 'Bit (GNAT, DEC Ada)
- 'Bit_Position (GNAT)
- 'Code_Address (GNAT)
- 'Default_Bit_Order (GNAT)

E – H

- 'Elaborated (GNAT)
- 'Elab_Body (GNAT)
- 'Elab_Spec (GNAT)
- 'Emax (GNAT)
- 'Enum_Rep (GNAT)
- 'Epsilon (GNAT)
- 'Fixed_Value (GNAT)
- 'Has_Access_Values (GNAT)

- 'Has_Discriminants (GNAT)

I – N

- 'Img (GNAT)
- 'Integer_Value (GNAT)
- 'Machine_Size (GNAT, DEC Ada)
- 'Max_Interrupt_Priority (GNAT)
- 'Max_Priority (GNAT)
- 'Maximum_Alignment (GNAT)
- 'Mechanism_Code (GNAT)
- 'Null_Parameter (GNAT, DEC Ada)

O – T

- 'Object_Size (GNAT)
- 'Passed_By_Reference (GNAT)
- 'Range_Length (GNAT)
- 'Storage_Unit (GNAT)
- 'Target_Name (GNAT)
- 'Tick (GNAT)
- 'To_Address (GNAT)
- 'Type_Class (GNAT, DEC Ada)

U – Z

- 'UET_Address (GNAT)
- 'Unconstrained_Array (GNAT)
- 'Universal_Literal_String (GNAT)
- 'Unrestricted_Access (GNAT)
- 'VADS_Size (GNAT)
- 'Value_Size (GNAT)
- 'Wchar_T_Size (GNAT)
- 'Word_Size (GNAT)

See also

Wikibook

- Ada Programming

Ada 95 Reference Manual

- 4.1.4 Attributes (Annotated)

- Annex K (informative) Language-Defined Attributes (Annotated)

Ada 2005 Reference Manual

- 4.1.4 Attributes (Annotated)
- Annex K (informative) Language-Defined Attributes (Annotated)

35 Pragmas

Description

Pragmas control the compiler, i.e. they are compiler directives. They have the standard form of

```
pragma Name (Parameter_List);
```

where the parameter list is optional.

List of language defined pragmas

Some pragmas are specially marked:

Ada 2005
: This is a new Ada 2005 pragma.

Obsolescent
: This is a deprecated pragma and it should not be used in new code.

A – H

- All_Calls_Remote
- Assert (Ada 2005)
- Assertion_Policy (Ada 2005)
- Asynchronous
- Atomic
- Atomic_Components
- Attach_Handler
- Controlled
- Convention
- Detect_Blocking (Ada 2005)
- Discard_Names
- Elaborate
- Elaborate_All
- Elaborate_Body
- Export

I – O

- Import
- Inline
- Inspection_Point
- Interface (Obsolescent)
- Interface_Name (Obsolescent)
- Interrupt_Handler
- Interrupt_Priority

- Linker_Options
- List
- Locking_Policy
- Memory_Size (Obsolescent)
- No_Return (Ada 2005)
- Normalize_Scalars
- Optimize

P – R

- Pack
- Page
- Partition_Elaboration_Policy (Ada 2005)
- Preelaborable_Initialization (Ada 2005)
- Preelaborate
- Priority
- Priority_Specific_Dispatching (Ada 2005)
- Profile (Ada 2005)
- Pure
- Queueing_Policy
- Relative_Deadline (Ada 2005)
- Remote_Call_Interface
- Remote_Types
- Restrictions
- Reviewable

S – Z

- Shared (Obsolescent)
- Shared_Passive
- Storage_Size
- Storage_Unit (Obsolescent)
- Suppress
- System_Name (Obsolescent)
- Task_Dispatching_Policy
- Unchecked_Union (Ada 2005)
- Unsuppress (Ada 2005)
- Volatile
- Volatile_Components

List of implementation defined pragmas

The following pragmas are not available in all Ada compilers, only in those that had implemented them.

You can help Wikibooks adding implementation dependent attributes of other compilers:

GNAT
: Implementation defined pragma of the GNAT compiler.

DEC Ada
: Implementation defined pragma of the DEC Ada compiler.

A – C

- Abort_Defer (GNAT)
- Ada_83 (GNAT)
- Ada_95 (GNAT)
- Ada_05 (GNAT)
- Annotate (GNAT)
- Ast_Entry (GNAT, DEC Ada)
- C_Pass_By_Copy (GNAT)
- Comment (GNAT)
- Common_Object (GNAT, DEC Ada)
- Compile_Time_Warning (GNAT)
- Complex_Representation (GNAT)
- Component_Alignment (GNAT, DEC Ada)
- Convention_Identifier (GNAT)
- CPP_Class (GNAT)
- CPP_Constructor (GNAT)
- CPP_Virtual (GNAT)
- CPP_Vtable (GNAT)

D – H

- Debug (GNAT)
- Elaboration_Checks (GNAT)
- Eliminate (GNAT)
- Export_Exception (GNAT, DEC Ada)
- Export_Function (GNAT, DEC Ada)
- Export_Object (GNAT, DEC Ada)
- Export_Procedure (GNAT, DEC Ada)
- Export_Value (GNAT)
- Export_Valued_Procedure (GNAT, DEC Ada)
- Extend_System (GNAT)
- External (GNAT)
- External_Name_Casing (GNAT)
- Finalize_Storage_Only (GNAT)
- Float_Representation (GNAT, DEC Ada)

I – L

- Ident (GNAT, DEC Ada)
- Import_Exception (GNAT, DEC Ada)
- Import_Function (GNAT, DEC Ada)

- Import_Object (GNAT, DEC Ada)
- Import_Procedure (GNAT, DEC Ada)
- Import_Valued_Procedure (GNAT, DEC Ada)
- Initialize_Scalars (GNAT)
- Inline_Always (GNAT)
- Inline_Generic (GNAT, DEC Ada)
- Interface_Name (GNAT, DEC Ada)
- Interrupt_State (GNAT)
- Keep_Names (GNAT)
- License (GNAT)
- Link_With (GNAT)
- Linker_Alias (GNAT)
- Linker_Section (GNAT)
- Long_Float (GNAT: OpenVMS, DEC Ada)

M – S

- Machine_Attribute (GNAT)
- Main_Storage (GNAT, DEC Ada)
- Obsolescent (GNAT)
- Passive (GNAT, DEC Ada)
- Polling (GNAT)
- Profile_Warnings (GNAT)
- Propagate_Exceptions (GNAT)
- Psect_Object (GNAT, DEC Ada)
- Pure_Function (GNAT)
- Restriction_Warnings (GNAT)
- Share_Generic (GNAT, DEC Ada)
- Source_File_Name (GNAT)
- Source_File_Name_Project (GNAT)
- Source_Reference (GNAT)
- Stream_Convert (GNAT)
- Style_Checks (GNAT)
- Subtitle (GNAT)
- Suppress_All (GNAT, DEC Ada)
- Suppress_Exception_Locations (GNAT)
- Suppress_Initialization (GNAT)

T – Z

- Task_Info (GNAT)
- Task_Name (GNAT)
- Task_Storage (GNAT, DEC Ada)
- Thread_Body (GNAT)
- Time_Slice (GNAT, DEC Ada)
- Title (GNAT, DEC Ada)
- Unimplemented_Unit (GNAT)
- Universal_Data (GNAT)
- Unreferenced (GNAT)

- Unreserve_All_Interrupts (GNAT)
- Use_VADS_Size (GNAT)
- Validity_Checks (GNAT)
- Warnings (GNAT)
- Weak_External (GNAT)

See also

Wikibook

- Ada Programming

Ada Reference Manual

Ada 95

- 2.8 Pragmas (Annotated)
- Annex L (informative) Language-Defined Pragmas (Annotated)

Ada 2005

- 2.8 Pragmas (Annotated)
- Annex L (informative) Language-Defined Pragmas (Annotated)

36 Libraries: Standard

The **Standard** package is implicit. This means two things:

1. You do not need to **with** or **use** the package, in fact you cannot (see below). It's always available (except where hidden by a homograph, RM 8.3 (8) (Annotated)).
2. **Standard** may contain constructs which are not quite legal Ada (like the definitions of `Character` and `Wide_Character`).

A **with** clause mentioning `Standard` references a user-defined package `Standard` that hides the predefined one. So do not do this. However any library unit hidden by a homograph can be made visible again by qualifying its name with `Standard`, like e.g. `Standard.My_Unit`.

Implementation

Since the package `Standard` is very important for portability, here are some examples for various compilers:

- The package Standard from ISO 8652.
- The package Standard from GNAT.
- The package Standard from Rational Apex.
- The package Standard from ObjectAda
- The Standard definitions for AppletMagic

Portability

The only mandatory types in Standard are Boolean, Integer and its subtypes, Float, Character, Wide_Character, String, Wide_String, Duration. There is an implementation permission in RM A.1 (51) (Annotated) that there may be more integer and floating point types and an implementation advice RM A.1 (52) (Annotated) about the names to be chosen. There even is no requirement that those additional types must have different sizes. So it is e.g. legal for an implementation to provide two types Long_Integer and Long_Long_Integer which both have the same range and size.

Note that the ranges and sizes of these types can be different in every platform (except of course for Boolean and [Wide_]Character). There is an implementation requirement that the size of type Integer is at least 16 bits, and that of Long_Integer at least 32 bits (if present) RM 3.5.4 (21..22) (Annotated). So if you want full portability of your types, do not use types from Standard (except where you must, see below), rather define you own types. A compiler will reject any type declaration whose range it cannot satisfy.

This means e.g. if you need a 64 bit type and find that with your current implementation Standard.Long_Long_Integer is such a type, when porting your program to another implementation, this type may be shorter, but the compiler will not tell you - and your program will most probably crash. However, when you define your own type like

```
type My_Integer_64 is range -(2**63) .. +(2**63 - 1);
```

then, when porting to an implementation that cannot satisfy this range, the compiler will reject your program.

The type Integer is mandatory when you use [wide] strings or exponentiation x**i. This is why some projects even define their own strings, but this means throwing out the child with the bath tub. Using Integer with strings and exponentiation will normally not lead to portability issues.

See also

Wikibook

- Ada Programming
- Ada Programming/Libraries#Predefined Language Libraries

Ada Reference Manual

- A.1 The Package Standard (Annotated)
- 3.5.4 Integer Types (Annotated)
- 3.5.7 Floating Point Types (Annotated)

Ada Quality and Style Guide

- 7.1.1 Obsolescent Features – Avoid using the package ASCII
- 7.2.1 Predefined Numeric Types – Avoid the predefined numerictypes

37 Libraries: Ada

The **Ada** package is only an anchor or namespace for Ada's standard library. Most compilers will not allow you to add new packages to the Ada hierarchy and even if your compiler allows it you should not do so since all packagenames starting with *Ada.* are reserved for future extensions.

List of language defined child units

The following library units (packages and generic subprograms) are descendents of the package Ada.

Ada 2005
This package is proposed for the Ada 2005 standard.

A – C

- Ada.Assertions (Ada 2005)
- Ada.Asynchronous_Task_Control
- Ada.Calendar
- Ada.Calendar.Arithmetic (Ada 2005)
- Ada.Calendar.Formatting (Ada 2005)
- Ada.Calendar.Time_Zones (Ada 2005)
- Ada.Characters
- Ada.Characters.Conversions (Ada 2005)
- Ada.Characters.Handling
- Ada.Characters.Latin_1
- Ada.Command_Line
- Ada.Complex_Text_IO (Ada 2005)
- Ada.Containers (Ada 2005)
- Ada.Containers.Doubly_Linked_Lists (Ada 2005)
- Ada.Containers.Generic_Array_Sort (Ada 2005 generic procedure)
- Ada.Containers.Generic_Constrained_Array_Sort (Ada 2005 generic procedure)
- Ada.Containers.Hashed_Maps (Ada 2005)
- Ada.Containers.Hashed_Sets (Ada 2005)
- Ada.Containers.Indefinite_Doubly_Linked_Lists (Ada 2005)
- Ada.Containers.Indefinite_Hashed_Maps (Ada 2005)
- Ada.Containers.Indefinite_Hashed_Sets (Ada 2005)
- Ada.Containers.Indefinite_Ordered_Maps (Ada 2005)
- Ada.Containers.Indefinite_Ordered_Sets (Ada 2005)
- Ada.Containers.Indefinite_Vectors (Ada 2005)
- Ada.Containers.Ordered_Maps (Ada 2005)
- Ada.Containers.Ordered_Sets (Ada 2005)
- Ada.Containers.Vectors (Ada 2005)

D – F

- Ada.Decimal
- Ada.Direct_IO

- Ada.Directories (Ada 2005)
- Ada.Directories.Information (Ada 2005)
- Ada.Dispatching (Ada 2005)
- Ada.Dispatching.EDF (Ada 2005)
- Ada.Dispatching.Round_Robin (Ada 2005)
- Ada.Dynamic_Priorities
- Ada.Environment_Variables (Ada 2005)
- Ada.Exceptions
- Ada.Execution_Time (Ada 2005)
- Ada.Execution_Time.Timers (Ada 2005)
- Ada.Execution_Time.Group_Budgets (Ada 2005)
- Ada.Finalization
- Ada.Float_Text_IO
- Ada.Float_Wide_Text_IO
- Ada.Float_Wide_Wide_Text_IO (Ada 2005)

G – R

- Ada.Integer_Text_IO
- Ada.Integer_Wide_Text_IO
- Ada.Integer_Wide_Wide_Text_IO (Ada 2005)
- Ada.Interrupts
- Ada.Interrupts.Names
- Ada.IO_Exceptions
- Ada.Numerics
- Ada.Numerics.Complex_Arrays (Ada 2005)
- Ada.Numerics.Complex_Elementary_Functions
- Ada.Numerics.Complex_Types
- Ada.Numerics.Discrete_Random
- Ada.Numerics.Elementary_Functions
- Ada.Numerics.Float_Random
- Ada.Numerics.Generic_Complex_Arrays (Ada 2005)
- Ada.Numerics.Generic_Complex_Elementary_Functions
- Ada.Numerics.Generic_Complex_Types
- Ada.Numerics.Generic_Elementary_Functions
- Ada.Numerics.Generic_Real_Arrays (Ada 2005)
- Ada.Numerics.Real_Arrays (Ada 2005)

R – S

- Ada.Real_Time
- Ada.Real_Time.Timing_Events (Ada 2005)
- Ada.Sequential_IO
- Ada.Storage_IO
- Ada.Streams
- Ada.Streams.Stream_IO
- Ada.Strings
- Ada.Strings.Bounded
- Ada.Strings.Bounded.Hash (Ada 2005 generic function)

- Ada.Strings.Fixed
- Ada.Strings.Fixed.Hash (Ada 2005 generic function)
- Ada.Strings.Hash (Ada 2005 generic function)
- Ada.Strings.Maps
- Ada.Strings.Maps.Constants
- Ada.Strings.Unbounded
- Ada.Strings.Unbounded.Hash (Ada 2005 generic function)
- Ada.Strings.Wide_Bounded
- Ada.Strings.Wide_Bounded.Wide_Hash (Ada 2005 generic function)
- Ada.Strings.Wide_Fixed
- Ada.Strings.Wide_Fixed.Wide_Hash (Ada 2005 generic function)
- Ada.Strings.Wide_Hash (Ada 2005 generic function)
- Ada.Strings.Wide_Maps
- Ada.Strings.Wide_Maps.Wide_Constants
- Ada.Strings.Wide_Unbounded
- Ada.Strings.Wide_Unbounded.Wide_Hash (Ada 2005 generic function)
- Ada.Strings.Wide_Wide_Bounded (Ada 2005)
- Ada.Strings.Wide_Wide_Bounded.Wide_Wide_Hash (Ada 2005 generic function)
- Ada.Strings.Wide_Wide_Fixed (Ada 2005)
- Ada.Strings.Wide_Wide_Fixed.Wide_Wide_Hash (Ada 2005 generic function)
- Ada.Strings.Wide_Wide_Hash (Ada 2005 generic function)
- Ada.Strings.Wide_Wide_Maps (Ada 2005)
- Ada.Strings.Wide_Wide_Maps.Wide_Wide_Constants (Ada 2005)
- Ada.Strings.Wide_Wide_Unbounded (Ada 2005)
- Ada.Strings.Wide_Wide_Unbounded.Wide_Wide_Hash (Ada 2005 generic function)

- Ada.Synchronous_Task_Control

T – U

- Ada.Tags
- Ada.Tags.Generic_Dispatching_Constructor (generic function)
- Ada.Task_Attributes
- Ada.Task_Identification
- Ada.Task_Termination (Ada 2005)
- Ada.Text_IO
- Ada.Text_IO.Bounded_IO (Ada 2005)
- Ada.Text_IO.Complex_IO
- Ada.Text_IO.Decimal_IO (Nested package of Ada.Text_IO)
- Ada.Text_IO.Editing
- Ada.Text_IO.Enumeration_IO (Nested package of Ada.Text_IO)
- Ada.Text_IO.Fixed_IO (Nested package of Ada.Text_IO)
- Ada.Text_IO.Float_IO (Nested package of Ada.Text_IO)
- Ada.Text_IO.Integer_IO (Nested package of Ada.Text_IO)
- Ada.Text_IO.Modular_IO (Nested package of Ada.Text_IO)
- Ada.Text_IO.Text_Streams
- Ada.Text_IO.Unbounded_IO (Ada 2005)
- Ada.Unchecked_Conversion (generic function)
- Ada.Unchecked_Deallocation (generic procedure)

W – Z

- Ada.Wide_Characters (Ada 2005)
- Ada.Wide_Text_IO
- Ada.Wide_Text_IO.Bounded_IO (Ada 2005)
- Ada.Wide_Text_IO.Complex_IO
- Ada.Wide_Text_IO.Decimal_IO (Nested package of Ada.Wide_Text_IO)
- Ada.Wide_Text_IO.Editing
- Ada.Wide_Text_IO.Enumeration_IO (Nested package of Ada.Wide_Text_IO)
- Ada.Wide_Text_IO.Fixed_IO (Nested package of Ada.Wide_Text_IO)
- Ada.Wide_Text_IO.Float_IO (Nested package of Ada.Wide_Text_IO)
- Ada.Wide_Text_IO.Integer_IO (Nested package of Ada.Wide_Text_IO)
- Ada.Wide_Text_IO.Modular_IO (Nested package of Ada.Wide_Text_IO)
- Ada.Wide_Text_IO.Text_Streams
- Ada.Wide_Text_IO.Unbounded_IO (Ada 2005)
- Ada.Wide_Wide_Characters (Ada 2005)
- Ada.Wide_Wide_Text_IO (Ada 2005)
- Ada.Wide_Wide_Text_IO.Bounded_IO (Ada 2005)
- Ada.Wide_Wide_Text_IO.Complex_IO (Ada 2005)
- Ada.Wide_Wide_Text_IO.Decimal_IO (Nested package of Ada.Wide_Wide_Text_IO)
- Ada.Wide_Wide_Text_IO.Editing (Ada 2005)
- Ada.Wide_Wide_Text_IO.Enumeration_IO (Nested package of Ada.Wide_Wide_Text_IO)
- Ada.Wide_Wide_Text_IO.Fixed_IO (Nested package of Ada.Wide_Wide_Text_IO)
- Ada.Wide_Wide_Text_IO.Float_IO (Nested package of Ada.Wide_Wide_Text_IO)
- Ada.Wide_Wide_Text_IO.Integer_IO (Nested package of Ada.Wide_Wide_Text_IO)
- Ada.Wide_Wide_Text_IO.Modular_IO (Nested package of Ada.Wide_Wide_Text_IO)
- Ada.Wide_Wide_Text_IO.Text_Streams (Ada 2005)
- Ada.Wide_Wide_Text_IO.Unbounded_IO (Ada 2005)

List of implementation defined child units

The Reference Manual allows compiler vendors to add extensions to the Standard Libraries. However, these extensions cannot be directly childs of the package Ada, only grandchilds -- for example Ada.Characters.Latin_9.

Currently, there are only listed the implementation defined attributes of the GNAT compiler. You can help Wikibooks adding implementation dependent attributes of other compilers:

GNAT
: Extended package implemented by GNAT.

ObjectAda
: Extended package implemented by ObjectAda.

APEX
: Extended package implemented by IBM/Rational APEX.

A – K

- Ada.Characters.Latin_9 (GNAT)

- Ada.Characters.Wide_Latin_1 (GNAT)
- Ada.Characters.Wide_Latin_9 (GNAT)
- Ada.Characters.Wide_Wide_Latin_1 (GNAT)
- Ada.Characters.Wide_Wide_Latin_9 (GNAT)
- Ada.Command_Line.Environment (GNAT)
- Ada.Command_Line.Remove (GNAT)
- Ada.Direct_IO.C_Streams (GNAT)
- Ada.Exceptions.Is_Null_Occurrence (GNAT child function)
- Ada.Exceptions.Traceback (GNAT)

L – Q

- Ada.Long_Float_Text_IO (GNAT)
- Ada.Long_Float_Wide_Text_IO (GNAT)
- Ada.Long_Integer_Text_IO (GNAT)
- Ada.Long_Integer_Wide_Text_IO (GNAT)
- Ada.Long_Long_Float_Text_IO (GNAT)
- Ada.Long_Long_Float_Wide_Text_IO (GNAT)
- Ada.Long_Long_Integer_Text_IO (GNAT)
- Ada.Long_Long_Integer_Wide_Text_IO (GNAT)
- Ada.Numerics.Long_Complex_Elementary_Functions (GNAT)
- Ada.Numerics.Long_Complex_Types (GNAT)
- Ada.Numerics.Long_Elementary_Functions (GNAT)
- Ada.Numerics.Long_Long_Complex_Elementary_Functions (GNAT)
- Ada.Numerics.Long_Long_Complex_Types (GNAT)
- Ada.Numerics.Long_Long_Elementary_Functions (GNAT)
- Ada.Numerics.Short_Complex_Elementary_Functions (GNAT)
- Ada.Numerics.Short_Complex_Types (GNAT)
- Ada.Numerics.Short_Elementary_Functions (GNAT)

R – Z

- Ada.Sequential_IO.C_Streams (GNAT)
- Ada.Short_Float_Text_IO (GNAT)
- Ada.Short_Float_Wide_Text_IO (GNAT)
- Ada.Short_Integer_Text_IO (GNAT)
- Ada.Short_Integer_Wide_Text_IO (GNAT)
- Ada.Short_Short_Integer_Text_IO (GNAT)
- Ada.Short_Short_Integer_Wide_Text_IO (GNAT)
- Ada.Streams.Stream_IO.C_Streams (GNAT)
- Ada.Strings.Unbounded.Text_IO (GNAT)
- Ada.Strings.Wide_Unbounded.Wide_Text_IO (GNAT)
- Ada.Strings.Wide_Wide_Unbounded.Wide_Wide_Text_IO (GNAT)
- Ada.Text_IO.C_Streams (GNAT)
- Ada.Wide_Text_IO.C_Streams (GNAT)
- Ada.Wide_Wide_Text_IO.C_Streams (GNAT)

See also

Wikibook

- Ada Programming
- Ada Programming/Libraries
- Ada Programming/Libraries/Standard
- Ada Programming/Libraries/System
- Ada Programming/Libraries/Interfaces

Ada Reference Manual

A.2 The Package Ada (Annotated)

38 Libraries: Interfaces

The Interfaces package helps in interfacing with other programming languages. Ada is one of the few languages where interfacing with other languages is part of the language standard. The language standard defines the interfaces for C, Cobol and Fortran. Of course any implementation might define further interfaces - GNAT for example defines an interface to C++.

Interfacing with other languages is actually provided by **pragma** Export, **pragma** Import and **pragma** Convention

Child Packages

- Interfaces.C
  - Interfaces.C.Extensions (GNAT)
- Interfaces.C.Pointers
- Interfaces.C.Streams (GNAT)
- Interfaces.C.Strings
- Interfaces.CPP (GNAT)
- Interfaces.COBOL
- Interfaces.Fortran
- Interfaces.OS2Lib (GNAT, OS/2)
  - Interfaces.OS2Lib.Errors (GNAT, OS/2)
- Interfaces.OS2Lib.Synchronization (GNAT, OS/2)
- Interfaces.OS2Lib.Threads (GNAT, OS/2)
- Interfaces.Packed_Decimal (GNAT)
- Interfaces.VxWorks (GNAT, VxWorks)
  - Interfaces.VxWorks.IO (GNAT, VxWorks)

See also

Wikibook

- Ada Programming
- Ada Programming/Libraries
- Ada Programming/Libraries/Standard
- Ada Programming/Libraries/Ada
- Ada Programming/Libraries/System

Ada Reference Manual

Ada 95

- Annex B Interface to Other Languages (Annotated)
- B.2 The Package Interfaces (Annotated)

Ada 2005

- Annex B Interface to Other Languages (Annotated)
- B.2 The Package Interfaces (Annotated)

39 LIBRARIES: SYSTEM

40 Libraries: GNAT

The GNAT package hierarchy defines several units for general purpose programming provided by the GNAT compiler. It is distributed along with the compiler and uses the same license.

GNAT-4-ObjectAda is a project for porting the GNAT library to the ObjectAda compiler

Child packages

- GNAT.Array_Split
- GNAT.AWK
- GNAT.Bounded_Buffers
- GNAT.Bounded_Mailboxes
- GNAT.Bubble_Sort
- GNAT.Bubble_Sort_A
- GNAT.Bubble_Sort_G
- GNAT.Calendar
  - GNAT.Calendar.Time_IO
- GNAT.Case_Util
- GNAT.CGI
  - GNAT.CGI.Cookie
- GNAT.CGI.Debug
- GNAT.Command_Line
- GNAT.Compiler_Version
- GNAT.CRC32
- GNAT.Ctrl_C
- GNAT.Current_Exception
- GNAT.Debug_Pools
- GNAT.Debug_Utilities
- GNAT.Directory_Operations
  - GNAT.Directory_Operations.Iteration
- GNAT.Dynamic_HTables
- GNAT.Dynamic_Tables
- GNAT.Exception_Actions
- GNAT.Exceptions
- GNAT.Exception_Traces
- GNAT.Expect
- GNAT.Float_Control
- GNAT.Heap_Sort
- GNAT.Heap_Sort_A
- GNAT.Heap_Sort_G
- GNAT.HTable
- GNAT.IO
- GNAT.IO_Aux
- GNAT.Lock_Files
- GNAT.MD5
- GNAT.Memory_Dump
- GNAT.Most_Recent_Exception
- GNAT.OS_Lib

- GNAT.Perfect_Hash_Generators
- GNAT.Regexp
- GNAT.Registry
- GNAT.Regpat
- GNAT.Secondary_Stack_Info
- GNAT.Semaphores
- GNAT.Signals
- GNAT.Sockets (GNAT.Sockets examples)
    - GNAT.Sockets.Constants
- GNAT.Sockets.Linker_Options
- GNAT.Sockets.Thin
- GNAT.Source_Info
- GNAT.Spelling_Checker
- GNAT.Spitbol
    - GNAT.Spitbol.Patterns
- GNAT.Spitbol.Table_Boolean new
- GNAT.Spitbol.Table_Integer
- GNAT.Spitbol.Table_VString new
- GNAT.Strings
- GNAT.String_Split
- GNAT.Table
- GNAT.Task_Lock
- GNAT.Threads
- GNAT.Traceback
    - GNAT.Traceback.Symbolic
- GNAT.Wide_String_Split

See also

External links

- The GNAT Library

Wikibook

- Ada Programming
- Ada Programming/Libraries

41 Libraries

Predefined Language Libraries

The library which comes with Ada in general and GNAT in particular. Ada's built in library is quite extensive and well structured. These chapters too are more reference like.

- Standard
- Ada
- Interfaces
- System
- GNAT

Other Language Libraries

Other libraries which arenot part of the standard but freelyavailable.

- Multi Purpose
- Container Libraries
- GUI Libraries
- Distributed Objects
- Database
- Web Programming
- Input/Output

See also

Wikibook

- Ada Programming

Ada Reference Manual

- Annex A (normative) Predefined Language Environment (Annotated)

42 Libraries: Multi-Purpose

AdaCL, Ada Class Library
: Filtering of text files, string tools, process control, command line parsing, CGI, garbage collector, components.

See also

Wikibook

- Ada Programming
- Ada Programming/Libraries

Ada Reference Manual

-- does not apply --

43 Libraries: Container

The following Libraries help you in store an manage object inside container classes:

Booch Components
: the most complete of all container class libraries (at least when used with AdaCL, Ada Class Library).

AdaCL, Ada Class Library
: A Booch Components extension pack for storing indefinite objects.

Charles
: Build on the C++ STL and therefore very easy to learn for C++ developers.

AI302
: Proof of concept for Ada.Containers

Ada.Containers
: *This language feature is only available in Ada 2005 standard.*

See also

Wikibook

- Ada Programming
- Ada Programming/Libraries

Ada Reference Manual

- A.18.1 The Package Containers (Annotated)

44 LIBRARIES: GUI

The following libraries can be used to make Graphical User Interfaces:

- GtkAda
- GWindows
- Qt4Ada

See also

Wikibook

- Ada Programming
- Ada Programming/Libraries

Ada Reference Manual

-- does not apply --

External Links

- adapower.com - Links to tools and Bindings for GUI Applications
- adapower.com - Examples of programming GUIs in Ada

45 Libraries: Distributed

The following Libraries help you in Distributed programming:

GLADE
: A full implementation of the Ada Annex E: Distributed Systems

PolyORB
: A CORBA and Annex E: Distributed Systems implementation.

See also

Wikibook

- Ada Programming
- Ada Programming/Libraries
- Programming:CORBA

Ada Reference Manual

- Annex E (normative) Distributed Systems (Annotated)

46 Libraries: Database

The following Libraries help you in Database programming:

GNADE
: description missing.

APQ
: Ada95 Binding to PostgreSQL and MySQL.

GWindows
: ODBC in Windows.

See also

Wikibook

- Ada Programming
- Ada Programming/Libraries

Ada Reference Manual

-- dowes not apply --

47 Libraries: Web

The following libraries help you in Internet or Web programming:

AdaCL, Ada Class Library
: Powerful GCI implementation.

XML/Ada
: XML and Unicode support.

AWS
: A full-featured Web-Server.

See also

Wikibook

- Ada Programming
- Ada Programming/Libraries

Ada Reference Manual

-- does not apply --

48 LIBRARIES: IO

The following libraries help you when doing input/output:

AdaCL, Ada Class Library
.
XML/Ada
XML and Unicode support.

See also

Wikibook

- Ada Programming
- Ada Programming/Libraries

Ada Reference Manual

-- does not apply --

49 Platform

Ada is known to be very portable, but there is sometimes a necessity of using a specific platform feature. For that matter, there are some non-standard libraries.

- Linux
- Windows
- POSIX systems
- Virtual machines
  - Java
- .NET

See also

Wikibook

- Ada Programming

Ada Reference Manual

-- does not apply --

Ada Quality and Style Guide

- Chapter 7: Portability

50 Platform: Linux

The following libraries help you when you target the Linux Platform.

Florist
: POSIX.5 binding. It will let you perform Linux system calls in the POSIX subset.

Ncurses
: text terminal library.

Texttools
: ncurses-based library for the Linux console.

GtkAda
: GUI library (actually multiplatform).

See also

Wikibook

- Ada Programming
- Ada Programming/Libraries

Ada Reference Manual

-- does not apply --

External resources

- The Big Online Book of Linux Ada Programming
- Ada in Debian GNU/Linux, slides suitable for a 50minute presentation, by Ludovic Brenta.

51 Platform: Windows

The following Libraries and Tools help you when you target the MS-Windows Platform.

GWindows
: Win32 binding

CLAW
: Another Win32 binding that works with any Ada 95 compiler. An introductory edition is available free of charge for non-commercial use.

GNATCOM
: COM/DCOM/ActiveX binding

GNAVI
: Visual RAD (Rapid application development) Development environment

Console
: Libraries for console I/O.

Visual C++ - GNAT interface
: Guide for calling Ada functions from C++ using GNAT and Visual C++.

See also

Wikibook

- Ada Programming
- Ada Programming/Libraries

Ada Reference Manual

-- does not apply --

52 Platform: VM

The following tools help you when you target a virtual machine.

Java
: Programming Ada 95 for Java's JVM (JGnat, AppletMagic)

.NET
: Programming Ada for the .NET Platform (A#)

See also

Wikibook

- Ada Programming
- Ada Programming/Libraries

Ada Reference Manual

-- does not apply --

53 Portals

AdaBasis
: AdaBasis consists of about 560 MB of public domain source code and documents, mainly taken from the Public Ada Library (PAL). The software has been classified and is presented in a hierarchical manner, separated in different application domains, and, for some domains, with an additional multi-faceted searching facility.

  The intent is to provide students, teachers and researchers with a large collection of reusable Ada components and systems for use in language and software engineering courses.

  AdaBasis was set up by the Programming Languages Group of the Institut für Informatik at the University of Stuttgart, Germany. They plan to enlarge the library in the future, and welcome free public domain contributions. For more informations or to make suggestions please contact adabasis@informatik.uni-stuttgart.de

AdaWorld
: Here you'll find all the different projects that are either hosted here or elsewhere on the web and talked about, documented and updated here. For the sake of organization, the projects have been grouped into five categories, which will also be broken down into sub categories where necessary.

AdaPower
: Ada Tools and Resources.

SourceForge
: Currently there are 102 Ada projects hosted at SourceForge — including the example programs for Ada Programming wikibook.

The Public Ada Library (PAL)
: The PAL is a library of Ada and VHDL software, information, and courseware that contains over 1 BILLION bytes of material (mainly in compressed form). All items in the PAL have been released to the public with unlimited distribution, and, in most cases (the exceptions are shareware), the items are freeware.

Ada and Software Engineering Library Version 2 (ASE2)
: *The ASE2 Library contains over 1.1GB of material on Ada and Software Engineering assembled through a collaboration with over 60 organizations*. Walnut Creek CDROM once sold copies of this library. Nowadays it is no longer maintained but is still hosted in the Ada Belgium FTP server. It may contain useful resources, but it is higly redundant with other libraries.

See also

Wikibook

- Ada Programming
- Ada Programming/Tutorials
- Ada Programming/Wikis

Ada Reference Manual

-- does not apply --

Ada Quality and Style Guide

-- does not apply --

54 GNU Free Documentation License

Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The

Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

External links

- GNU Free Documentation License (Wikipedia article on the license)
- Official GNU FDL webpage