WIKIBOOKS
Open books for an open world

WIKIBOOKS

Ada Programming/All Chapters

Preface

Ada. Time-tested, safe and secure.

Welcome to the **Ada Programming** tutorial at Wikibooks. This is the first Ada tutorial covering the Ada 2005, 2012 and 2022 standards. 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 "". At this date, there are more than 600 pages in this book, which makes **Ada Programming** one of the largest 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

Ada is a programming language suitable for all development needs. It 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.

Ada is named after Augusta Ada King-Noel, Countess of Lovelace.


Augusta Ada King, Countess of Lovelace.

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 faults before they become errors.
- **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, following 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 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 1970s. The first standard was published in 1980, just 3 years after design commenced. Ada compilers all support the same language; the only dialect, SPARK, is merely an annotated subset and can be compiled with an Ada compiler.

Consequences of these qualities are superior **reliability**, **reusability** and **maintainability**. For example, compared to programs written in C, programs written in Ada 83 contain *"70% fewer internal fixes and 90% 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 about 80% of the total cost of development. With support for object-oriented programming, Ada 95 may bring even more cost-benefit, depending on how objects are used; 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 applies 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, defence, medical, railroad, and nuclear industries.

The Language Reference Manual

The Ada Reference Manual (RM) is the official language definition. If you have a problem and no one else can help, 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 to the appropriate pages in the RM.

This tutorial covers *Ada Reference Manual — ISO/IEC 8652:2023 Language and Standard Libraries*, colloquially known as *Ada 2022* or just *Ada*.

You can browse the complete Reference Manual at http://www.ada-auth.org/standards/22rm/html/RM-TOC.html

There are two companion documents:

- The Annotated Reference Manual (http://www.adaic.org/resources/add_content/standards/22aarm/html/AA-TTL.html), an extended version of the RM aimed at compiler writers or other persons who want to know the fine details of the language.
- The Overview of Ada 2022 (http://www.ada-auth.org/standards/overview22.html), an explanation of the features of this language edition.

The Ada Information Clearinghouse (http://www.adaic.com/standards) also offers the older Ada 83, 95, 2005 and 2012 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 (http://www.ada-auth.org/) (the Ada RM explains how to do this, see http://www.ada-auth.org/standards/22rm/html/RM-0-2.html Introduction (58/1) ff). On this site, you can also see the list of "Ada Issues" raised by other people.

Ada Conformity Assessment Test Suite

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. The Ada standard does however allow compiler writers to include additional features and libraries that are not part of the standard.

Programming in Ada

Getting Started

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

- Basic Ada — **Read Me First!**
- Finding and Installing Ada
- Building an Ada program
- Ada Development Environment

Language Features

These chapters look at the broader picture, introducing you to the main Ada features in a tutorial style.

- Expressions
- Control Structures
- Type System
- Constants
- Representation Clauses
- Strings
- Subprograms

- Packages
- Input Output
- Exceptions
- Generics
- Tasking
- Object Orientation
- Contract Based Programming
- Memory Management (Access Types)
- New in Ada 2005
- New in Ada 2012
- New in Ada 2022
- Containers
- Interfacing to other Languages
- Coding Standards
- Ada Programming Tips
- Common Programming Errors

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

- Variables

Language Reference

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
  - Keywords
  - Delimiters
- Operators
- Attributes
- Aspects
- Pragmas
  - Restrictions

Predefined Language Libraries

This section is a reference of the Ada Standard Library, which is extensive and well structured. It has these four root packages:

- Standard
- Ada
- Interfaces
- System

Besides the Standard Library, compilers usually come with a built-in library. This chapter describes the GNAT library in particular.

- GNAT

External Libraries

This section is a reference of third-party Ada libraries which are not part of the compiler predefined environment but are freely available.

- Libraries

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

External resources

- Open-source portals
- Web Tutorials
- Web 2.0

Collections

Printable Versions

The following are collection pages. All collection pages are comprised of groups of the already available pages. You can use them for printing or to gain a quick overview. Please note that those pages are partly very long.

Tutorial
Show HTML (1,839 kb) — Download PDF (https://upload.wikimedia.org/wikipedia/commons/8/8d/Ada_Programming.pdf) (2,663 kb, 243 pages)
Keywords
Show HTML (470 kb) — Download PDF (https://upload.wikimedia.org/wikipedia/commons/6/67/Ada_Programming_Keywords.pdf) (290 kb, 59 pages)
Operators

Show HTML (232 kb) — Download PDF (https://upload.wikimedia.org/wikipedia/commons/6/65/Ada_Programming_Operators.pdf) (189 kb, 27 pages)

Source Code

The Source from the Book is available for download (https://sourceforge.net/project/showfiles.php?group_id=124904) and online browsing (http://wikibook-ada.sourceforge.net/html/index.html). 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. See Category:Book:Ada Programming or /All Chapters
2. Gaetan Allaert, Dirk Craeynest, Philippe Waroquiers (2003). "European air traffic flow management: porting a large application to GNU/linux". *Proceedings of the 2003 annual ACM SIGAda international conference on Ada*. SIGAda'03. pp. 29–37. doi:10.1145/958420.958426. ISBN 1-58113-476-2. http://www.sigada.org/conf/sigada2003/SIGAda2003-CDROM/SIGAda2003-Proceedings/p29-allaert.pdf. Retrieved 2009-01-02. Paper by Eurocontrol (PDF, 160 kB) on portability.
3. Stephen F. Zeigler (1995-03-30). "Comparing Development Costs of C and Ada". Retrieved 2009-01-02. "Our data indicates that Ada has saved us millions of development dollars."

Further reading

Ada 2005 textbooks

- John Barnes (2006). *Programming in Ada 2005*. Addison Wesley. ISBN 0-321-34078-7.
- Mordechai Ben-Ari (2009). *Ada for Software Engineers (Second Edition with Ada 2005)*. Springer. ISBN 978-1-84882-313-6.
- Alan Burns, Andy Wellings (2007). *Concurrent and Real-Time Programming in Ada*. Cambridge University Press. ISBN 978-0-521-86697-2.
- Nell Dale, John W. McCormick (2007). *Ada Plus Data Structures: An Object Oriented Approach* (2nd ed.). Jones and Bartlett. ISBN 0-7637-3794-1.

- John W. McCormick, Frank Singhoff, Jérôme Hugues (2011). *Building Parallel, Embedded, and Real-Time Applications with Ada*. Cambridge University Press. ISBN 978-0-521-19716-8.

Ada 2012 textbooks

- John Barnes (2014). *Programming in Ada 2012*. Cambridge University Press. ISBN 978-1-107-42481-4.
- Andrew T.Shvets (2020). *Beginning Ada Programming: From Novice to Professional*. Apress Media LLC, A Subsidiary of Springer Nature. ISBN 978-1-4842-5427-1.

Ada 2022 textbooks

- John Barnes (2022). Programming in Ada 2012 with a Preview of Ada 2022 (https://www.cambridge.org/us/universitypress/subjects/computer-science/software-engineering-and-development/programming-ada-2012-preview-ada-2022-2nd-edition?format=PB&isbn=9781009181341). Cambridge University Press. ISBN 9781009181341
- John Barnes (2024) [PLANNED]. Programming in Ada 2022 (https://www.cambridge.org/us/universitypress/subjects/computer-science/programming-languages-and-applied-logic/programming-ada-2022). Cambridge University Press. ISBN 9781009564779

Manuals and guides

- Ada Quality & Style Guide: Guidelines for Professional Programmers (wikibook)
- Rationale for Ada 2005 (http://www.adaic.com/standards/05rat/html/Rat-TTL.html), by John Barnes (2007)
- Ada 2005 Reference Manual (http://www.adaic.com/standards/05rm/html/RM-TTL.html) (2007)
- Ada Reference Card (https://github.com/bracke/AdaReferenceCard/releases) (in PDF format)

High-Integrity Software

- ISO/IEC TR 15942:2000, *Guide for the use of the Ada programming language in high integrity systems (http://www.dit.upm.es/~str/ork/documents/adahis.pdf)*. ISO Freely Available Standards [3] (http://standards.iso.org/ittf/PubliclyAvailableStandards/)
- ISO/IEC TR 24718:2005, *Guide for the use of the Ada Ravenscar Profile in high integrity systems (http://www.sigada.org/ada_letters/jun2004/ravenscar_article.pdf)*. ISO Freely Available Standards [4] (http://standards.iso.org/ittf/PubliclyAvailableStandards/)

- John Barnes (2003). *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley. ISBN 0-321-13616-0.

External links

Resources

- Ada Information Clearinghouse (http://www.adaic.org/) — News and resources
- comp.lang.ada (news://comp.lang.ada) (Web archive (https://usenet.ada-lang.io/comp.lang.ada/), Google groups (https://groups.google.com/group/comp.lang.ada)) — International Usenet newsgroup
- ada-lang.io (https://ada-lang.io) — Community site and forum

Research and user groups

- Journals:
  - *Ada-Letters (http://www.sigada.org/ada_letters/)*
  - *Ada User Journal (http://www.ada-europe.org/auj/)* (archive (http://www.ada-europe.org/auj/archive/))
- International Conferences/Workshops:
  - International Real-Time Ada Workshop (IRTAW) (http://portal.acm.org/browse_dl.cfm?linked=1&part=series&idx=SERIES176&coll=portal&dl=ACM) [IRTAW 15 (http://www.ctr.unican.es/irtaw-15/cfp.html)]
  - ACM SIGAda international conference on Ada (http://portal.acm.org/toc.cfm?id=SERIES332) [HILT 2012 (http://www.sigada.org/conf/hilt2012/)]
  - 18th International Conference on Reliable Software Technologies — Ada-Europe 2013 (http://www.ada-europe2013.org/)
  - 17th International Conference on Reliable Software Technologies — Ada-Europe 2012 (http://www.cister.isep.ipp.pt/ae2012/)
  - Ada Connection 2011 (http://conferences.ncl.ac.uk/adaconnection2011/) (videos (http://www.adacore.com/home/ada_answers/lectures/ada-connection-2011/))
  - Ada "Developers Room" at FOSDEM (https://archive.fosdem.org/2020/schedule/track/ada/) (2020 (https://people.cs.kuleuven.be/~dirk.craeynest/ada-belgium/events/20/200201-fosdem.html))
- Local conferences:
  - Ada-Belgium Conference (http://www.cs.kuleuven.ac.be/~dirk/ada-belgium/events/local.html)

Associations

- ACM SIGAda (http://www.acm.org/sigada/) — ACM Special Interest Group on Ada
- Ada-Europe (http://www.ada-europe.org/)
- Ada Germany (http://www.ada-deutschland.de/)
- Ada Spain (http://www.adaspain.org/)
- Ada-Belgium (http://www.cs.kuleuven.be/~dirk/ada-belgium/)
- Ada-France (http://www.ada-france.org/)
- Ada Switzerland (http://www.ada-switzerland.ch/)

Free online books/courses

- learn.adacore.com (https://learn.adacore.com/)
- *The Big Online Book of Linux Ada Programming (http://www.pegasoft.ca/resources/boblap/book.html)*
- *Ada Distilled (http://www.sigada.org/education/pages/Ada-Distilled-07-27-2003-Color-Version.pdf)*

- *Ada in Action (http://www.cs.kuleuven.be/~dirk/ada-belgium/aia/contents.html)*
- *Introducing Ada 95 (http://www.seas.gwu.edu/~adagroup/sigada-website/barnes-html/intro.html)*
- *Learn Ada on the Web (http://web.archive.org/web/19980131094124/www.scism.sbu.ac.uk/law/lawhp.html)*
- *Quick Ada (http://goanna.cs.rmit.edu.au/~dale/ada/aln.html)*
- *Ada 95: The Craft of Object-Oriented Programming (http://archive.adaic.com/docs/craft/craft.html)* — Free textbook originally published by Prentice Hall
- Online Ada books (https://www.onlineprogrammingbooks.com/free-ada-books/)

Authors and contributors

This Wikibook has been written by:

- Martin Krischik (Contributions)
- Manuel Gómez (Contributions)
- Santiago Urueña (Contributions)
- C.K.W. Grein (Contributions, more and more 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)
- Mateus de Lima Oliveira (Contributions)
- Ed Falis
- Pascal Obry
- Bent Bracke (Contributions)

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

Basic Ada

"Hello, world!" programs

"Hello, world!"

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

File: hello_world_1.adb, Crate: basic (view (https://sourceforge.net/p/wikibook-ada/git/ci/develop/tree/wikibook/basic/src/hello_world_1.adb), plain text (https://sourceforge.net/p/wikibook-ada/git/ci/develop/tree/wikibook/basic/src/hello_world_1.adb?format=raw), download with Alire (https://alire.ada.dev/crates/wikibook), Alire crate info (https://wikibook-ada.sourceforge.net/samples/basic))

```
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. [1]

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.[2] This reduces the typing involved while still keeping some of the readability.

File: hello_world_2.adb, Crate: basic (view (https://sourceforge.net/p/wikibook-ada/git/ci/develop/tree/wikibook/basic/src/hello_world_2.adb), plain text (https://sourceforge.net/p/wikibook-ada/git/ci/develop/tree/wikibook/basic/src/hello_world_2.adb?format=raw), download with Alire (https://alire.ada.dev/crates/wikibook), Alire crate info (https://wikibook-ada.sourceforge.net/samples/basic))

```
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 local use

The `use` clause makes all the content of a package directly visible for the scope it is declared it. `use` can be placed locally or globally (see below). Like rename this reduces the typing involved while still keeping some of the readability.

File: hello_world_3.adb, Crate: basic (view (https://sourceforge.net/p/wikibook-ada/git/ci/develop/tree/wikibook/basic/src/hello_world_3.adb), plain text (https://sourceforge.net/p/wikibook-ada/git/ci/develop/tree/wikibook/basic/src/hello_world_3.adb?format=raw), download with Alire (https://alire.ada.dev/crates/wikibook), Alire crate info (https://wikibook-ada.sourceforge.net/samples/basic))

```
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.

"Hello, world!" with global use

Using `use` clause outside any scope will makes all the content of a package directly visible for the whole compilation unit. It allows even less typing but removes more of the readability and can lead to name clashes.

File: hello_world_4.adb, Crate: basic (view (https://sourceforge.net/p/wikibook-ada/git/ci/develop/tree/wikibook/basic/src/hello_world_4.adb), plain text (https://sourceforge.net/p/wikibook-ada/git/ci/develop/tree/wikibook/basic/src/hello_world_4.adb?format=raw), download with Alire (https://alire.ada.dev/crates/wikibook), Alire crate info (https://wikibook-ada.sourceforge.net/samples/basic))

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

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

With that many options one need to consider which option to use when. One suggested "rule of thumb":

- global `use` for the most used package(s)
- `renames` for most other package(s)
- local `use` for packages only used in a single procedure
- no use `use` or `renames` for package(s) only used once

You might have another simpler rule (for example, always `use` package Ada and its children, never `use` anything else).

Another rule from the early days of Ada development was global `use` for all packages unless a name clash occurs.

Compiling the "Hello, world!" program

For information on how to build the "Hello, world!" program on various compilers, see the Building chapter.

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 ensure a system that works "out of the box" and thus does not frighten away potential new users with unneeded complexity.

The GNAT project files, which you can download (https://sourceforge.net/project/showfiles.php?group_id=124904) 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 for a 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;
```

```
while Boolean expression loop
   statements
end loop;
```

```
for variable in range loop
   statements
```

```
end loop;
```

```
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;
```

```
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;
```

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.

Where to ask for help

Most Ada experts lurk on the Usenet newsgroups *comp.lang.ada* (English) and *fr.comp.lang.ada* (French); they are accessible either with a newsreader or through one of the many web interfaces. This is the place for all questions related to Ada.

People on these newsgroups are willing to help but will *not* do students' homework for them; they will not post complete answers to assignments. Instead, they will provide guidance for students to find their own answers.

For more online resources, see the External links section in this wikibook's introduction.

Notes

1. 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) (http://www.ada-auth.org/standards/12rm/html/RM-10-2.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-10-2.html)]: "…, 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.
2. `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`.

Installing

Ada compilers are available from several vendors, on a variety of host and target platforms. The Ada Resource Association (http://www.adaic.com) maintains a list of available compilers (http://www.adaic.com/compilers/comp-tool.html).

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

AdaMagic from SofCheck

SofCheck used to produce 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 used to offer:

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

SofCheck has merged (http://www.adacore.com/press/adacore-sofcheck-merge) with AdaCore under the AdaCore name, leaving no visible trace of AdaMagic offering on AdaCore website.

However, MapuSoft is now licensed to resell AdaMagic. They renamed it to "Ada-to-C/C++ changer (https://www.mapusoft.com/ada-to-c-changer/)". New name sounds like fake. Almost no Ada developer heard of MapuSoft. MapuSoft is never seen making Ada libraries, commercial or FLOSS. They are never seen at Ada conferences. Yet this is a real stuff, a validated Ada compiler that knows lots of tricks required to work on top of C/C++ compilers. E.g. it contains a proven knowledge of handling integer overflow with a special "-1" case.

Thanks to MapuSoft, AdaMagic really became available to developers. Get AppCOE (https://connect.mapusoft.com/demo_evalform.html), but not Win64 one, install it. In the MapuSoft/AppCOE_x32/Tools/Ada there will be AdaMagic. AdaMagic is known to support Win64, but AppCOE for Win64 is known to contain no AdaMagic at all.

Using AdaMagic from command line is badly supported in AppCOE, but possible. Set up ADA_MAGIC environment variable, edit Tools/Ada/{linux|windows}/SITE/rts_path to point to real path, edit SITE/config to get rid of unsupported C compiler keys, and compile via e.g.

```
adareg -key=`test_key | sed -e '/md5/!d;s/md5 = //'` Hello_World.adb
adabgen -key=`test_key | sed -e '/md5/!d;s/md5 = //'` Hello_World
```

Commercial; proprietary.

AdaMULTI from Green Hills Software

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

Languages supported	Ada 83, Ada 95, C, C++, Fortran
License for the run-time library	Proprietary, royalty free.
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, including x86, PowerPC, ARM, MIPS and ColdFire/68k. Safety-critical GMART and GSTART run-time libraries certified to DO-178B level A.
Available from	http://www.ghs.com/
Support	Commercial
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 (http://www.eembc.com) 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 two most recent Ada standards (2005 and 2012) but they do continue to actively market and develop their existing Ada products.

DEC Ada from HP

DEC Ada was an Ada 83 compiler for OpenVMS. While “DEC Ada” is probably the name most users know, the compiler has also been called “HP Ada”, "VAX Ada", and "Compaq Ada".

- Ada for OpenVMS Alpha Installation Guide (https://www.vmssoftware.com/pdfs/HP_branded_docs_1st_batch/ada_avms_ig.pdf) (PDF)
- Ada for OpenVMS VAX Installation Guide (https://www.vmssoftware.com/pdfs/HP_branded_docs_1st_batch/ada_vvms_ig.pdf) (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 (http://www.adacore.com) 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.

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 (or Community) Edition

As of May 2022, AdaCore no longer supports GNAT GPL. The recommended way (https://alire.ada.dev/transition%20from%20gnat%20community.html) to install all the tools and libraries that the Community Edition included is to use Alire (https://alire.ada.dev/), a package manager for Ada sources, which also provides toolchains (https://alire.ada.dev/docs/#toolchain-management). Although you can still download and install the last GNAT Community Edition that was published, there won't be any further release.

GNAT Community Edition 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.

As of GNAT GPL Edition 2013:

Languages supported	Ada 83, Ada 95, Ada 2005, Ada 2012, C, C++
License for the run-time library	pure GPL
Native platforms	GNU/Linux on x86_64; Microsoft Windows on i386; ; Mac OS X (Darwin, x86_64). Earlier releases have supported Solaris on SPARC, GNU/Linux on i386, Microsoft .NET on i386
Cross platforms	AVR, hosted on Windows; Java VM, hosted on Windows; Mindstorms NXT, hosted on Windows; ARM, hosted on Windows and Linux;
Compiler back-end	GCC 4.9
Available from	https://www.adacore.com/download
Support	None
Add-ons included	GDB, GPS 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/
Support	None
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, Ada 2012, C, and optionally C++
License for the run-time library	GNAT-modified GPL
Native platforms	many, see http://www.adacore.com/home/products/gnatpro/supported_platforms/
Cross platforms	many, see http://www.adacore.com/home/products/gnatpro/supported_platforms/; even more on request
Compiler back-end	GCC 4.3
Available from	http://www.adacore.com/ by subscription (commercial)
Support	Commercial; customer-only bug database
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 the Free Software Foundation (http://www.fsf.org/)'s GCC (http://gcc.gnu.org/) since October 2001. The Free Software Foundation does not distribute binaries, only sources. Its licensing of the run-time library for Ada (and other languages) allows the development of proprietary software without necessarily imposing the terms of the GPL.

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 later, or one of the releases from AdaCore (http://www.adacore.com) (3.15p, GPL Edition or Pro).

Since October 2003, AdaCore merge most of their changes from GNAT Pro into GCC during Stage 1 (http://gcc.gnu.org/develop.html#stage1); this happens once for each major release. Since GCC 3.4, AdaCore has gradually added support for revised language standards, first Ada 2005 and now Ada 2012.

GCC version 4.4 switched to version 3 of the GNU General Public License (http://www.gnu.org/licenses/gpl.html) and grants a Runtime Library Exception (http://www.gnu.org/licenses/gcc-exception.html) similar in spirit to the GNAT Modified General Public License used in all previous versions. This Runtime Library Exception applies to run-time libraries for all languages, not just Ada.

As of GCC 4.7, released on 2012-03-22:

Languages supported	Ada 83, Ada 95, Ada 2005, parts of Ada 2012, C, C++, Fortran 95, Java, Objective-C, Objective-C++ (and others)
License for the run-time library	GPL version 3 (http://www.gnu.org/licenses/gpl.html) with Runtime Library Exception (http://www.gnu.org/licenses/gcc-exception.html)
Native platforms	none (source only)
Cross platforms	none (source only)
Compiler back-end	GCC 4.7
Available from	http://gcc.gnu.org/ in source only form.
Support	Volunteer; public bug database
Add-ons included	none

The GNU Ada Project

The GNU Ada Project (http://gnuada.sourceforge.net) 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; there are instructions for building your own GNAT compiler for GNU/Linux (http://ada.krischik.com/index.php/Articles/CompileGNAT) and Mac OS X (http://forward-in-code.blogspot.com/2011/11/building-gcc-again.html) users.

Both GPL and GMGPL or GCC Runtime Library Exception (http://www.gnu.org/licenses/gcc-exception.html) versions of GNAT are available.

Languages supported	Ada 83, Ada 95, Ada 2005, C. (Some distributions also support Ada 2012, Fortran 90, Java, Objective C and Objective C++)
License for the run-time library	pure, GNAT-modified GPL, or GCC Runtime Library Exception
Native platforms	Fedora Core 4 and 5, MS-DOS, OS/2, Solaris 10, SuSE 10, Mac OS X, (more?)
Cross platforms	none
Compiler back-end	GCC 2.8.1, 3.4, 4.0, 4.1, 4.2, 4.3, 4.4, 4.5, 4.6 (various binary packages)
Available from	Sourceforge (http://sourceforge.net/projects/gnuada/files/)
Support	Volunteer; public bug database
Add-ons included	AdaBrowse, ASIS, Booch Components, Charles, GPS, GtkAda (more?)

A# (A-Sharp, a.k.a. Ada for .NET)

This compiler is historical as it has now been merged into GNAT GPL Edition and GNAT Pro.

A# is a port of Ada to the .NET Platform (http://www.microsoft.com/net/%7CMicrosoft). A# was originally developed at the Department of Computer Science at the United States Air Force Academy which distribute A# as a service to the Ada community under the terms of the GNU general public license. A# integrates well with Microsoft Visual Studio 2005, AdaGIDE and the RAPID open-source GUI Design tool. As of 2006-06-06:

Languages supported	Ada 83, Ada 95, C
License for the run-time library	pure GPL
Native platforms	Microsoft .NET
Cross platforms	none
Compiler back-end	GCC 3.4 (GNAT GPL 2006 Edition?)
Available from	http://sourceforge.net/projects/asharp/
Support	None (but see GNAT Pro)
Add-ons included	none.

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. As of Version 1.1.0 (2010-02-25):

Languages supported	Ada 83, Ada 95, Ada 2005, C
License for the run-time library	GNAT-Modified GPL
Host platforms	GNU/Linux and Microsoft Windows on i386
Target platforms	Various AVR 8-bit microcontrollers
Compiler back-end	GCC 4.7
Available from	http://avr-ada.sourceforge.net/
Support	Volunteer; public bug database
Add-ons included	partial Ada run time system, AVR peripherals support library

GNAT for LEON

The Real-Time Research Group of the Technical University of Madrid (UPM, *Universidad Politécnica de Madrid*) wrote a Ravenscar-compliant real-time kernel for execution on LEON processors and a modified run-time library. They also provide a GNAT cross-compiler. As of version 2.0.1:

Languages supported	Ada 83, Ada 95, Ada 2005, C
License for the run-time library	pure GPL
Native platforms	none
Cross platforms	GNU/Linux on i686 to LEON2 bare boards
Compiler back-end	GCC 4.1 (GNAT GPL 2007 Edition)
Available from	http://www.dit.upm.es/ork/
Support	?
Add-ons included	OpenRavenscar real-time kernel; minimal run-time library

GNAT for Macintosh (Mac OS X)

GNAT for Macintosh (http://www.macada.org/) provides both FSF (GMGPL) and AdaCore (GPL) versions of GNAT with Xcode and Carbon integration and bindings.

Note that this site was last updated for GCC 4.3 and Mac OS X Leopard (both PowerPC and Intel-based). Aside from the work on integration with Apple's Carbon graphical user interface and with Xcode 3.1 it may be preferable to see above.

There is also support at MacPorts (https://trac.macports.org/browser/trunk/dports/lang/gnat-gcc); the last update (at 25 Nov 2011) was for GCC 4.4.2.

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 (https://stef.genesix.org/aide/aide.html) 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 (just unzip to default c:\aide and run). 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	https://stef.genesix.org/aide/aide.html
Support	stef@genesix.org
Add-ons included	ASIS, GDB, GPS, GtkAda, Texinfo (more?)

Cygwin (for Microsoft Windows)

Cygwin (http://www.cygwin.com), the Linux-like environment for Windows, also contains a version of 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 (http://people.debian.org/~lbrenta/debian-ada-policy.html) which tries to make Debian the best Ada development *and deployment* platform. The development platform includes the compiler and many libraries, pre-packaged and integrated so as to be easy to use in any program. The deployment platform is the renowned *stable (http://www.debian.org/releases/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:

```
aptitude 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.

In the table below, the information about the future Debian 8.0 *Jessie* is accurate as of October 2014 and will change.

	3.1 *Sarge*	4.0 *Etch*	5.0 *Lenny*	6.0 *Squeeze*	7.0 *Wheezy*	8.0 *Jessie*
Release date	June 2005	April 2007	February 2009	February 2011	May 2013	April 2015
Languages supported	Ada 83, Ada 95, C	+Ada 2005, parts of Ada 2012, C, C++, Fortran 95, Java, Objective-C, Objective-C++				+Ada 2012
License for the run-time library	GNAT-modified GPL (both ZCX and SJLJ versions starting from 5.0 *Lenny*)			GPL version 3 with Run-time library exception		
Native platforms:	**3.1 *Sarge***	**4.0 *Etch***	**5.0 *Lenny***	**6.0 *Squeeze***	**7.0 *Wheezy***	**8.0 *Jessie***
alpha		yes	yes			
amd64		yes	yes	yes	yes	yes
armel				preliminary	yes	yes
armhf					yes	yes
hppa		yes	yes	yes		
hurd-i386					yes	yes
i386	yes	yes	yes	yes	yes	yes
ia64		yes	yes	yes	yes	
kfreebsd-amd64				yes	yes	yes
kfreebsd-i386		yes	yes	yes	yes	yes
mips		yes	yes	yes	yes	yes
mipsel		yes	yes	yes	yes	yes
powerpc	yes	yes	yes	yes	yes	yes
ppc64			yes	yes	yes	yes
s390		yes	yes	yes	yes	s390x
sparc	yes	yes	yes	yes	yes	yes
Cross platforms	none					
Compiler back-end	**GCC 2.8.1**	**GCC 4.1**	**GCC 4.3**	**GCC 4.4**	**GCC 4.6**	**GCC 4.9**
Available from	http://www.debian.org/					
Support	Volunteer; public bug database; paid support available from third parties; public mailing list (http://lists.debian.org/debian-ada)					
Add-ons included	**3.1 *Sarge***	**4.0 *Etch***	**5.0 *Lenny***	**6.0 *Squeeze***	**7.0 *Wheezy***	**8.0 *Jessie***
ada-reference-manual	1995	1995	1995	2005	2012	2012

AdaBindX	0.7.2					
AdaBrowse	4.0.2	4.0.2	4.0.2	4.0.3	4.0.3	-
AdaCGI	1.6	1.6	1.6	1.6	1.6	1.6
AdaControl		1.6r8	1.9r4	1.12r3	1.12r3	1.16r11
APQ (with PostgreSQL)				3.0	3.2	3.2
AdaSockets	1.8.4.7	1.8.4.7	1.8.4.7	1.8.8	1.8.10	1.8.11
Ahven			1.2	1.7	2.1	2.4
Alog			0.1	0.3	0.4.1	-
anet					0.1	0.3.1
ASIS	3.15p	2005	2007	2008	2010	2014
AUnit	1.01	1.03	1.03	1.03	1.03	3.7.1
AWS	2.0	2.2	2.5 prerelease	2.7	2.10.2	3.2.0
Charles	2005-02-17	(superseded by Ada.Containers in gnat)				
Florist	3.15p	2006	2006	2009	2011	2014
GDB	5.3	6.4	6.8	7.0.1	7.4.1	7.7.1
GLADE	3.15p	2006		(superseded by PolyORB)		
GMPAda				0.0.20091124	0.0.20120331	0.0.20131223
GNADE	1.5.1	1.6.1	1.6.1	1.6.2	1.6.2	-
GNAT Checker	1999-05-19	(superseded by AdaControl)				
GPRBuild				1.3.0w	2011	2014
GPS	2.1	4.0.1	4.0.1	4.3	5.0	5.3
GtkAda	2.4	2.8.1	2.8.1	2.14.2	2.24.1	2.24.4
Log4Ada				1.0	1.2	1.2
Narval				1.10.2		
OpenToken	3.0b	3.0b	3.0b	4.0b	4.0b	5.0a
PC/SC Ada				0.6	0.7.1	0.7.2

PolyORB				2.6 prerelease	2.8 prerelease	2.11 prerelease
PLPlot			5.9.0	5.9.5	5.9.5	5.10.0
Templates Parser		10.0+20060522	11.1	11.5	11.6	11.8
TextTools	2.0.3	2.0.3	2.0.5	2.0.6		2.1.0
XML/Ada	1.0	2.2	3.0	3.2	4.1	4.4
XML-EZ-out				1.06	1.06.1	1.06.1

The ADT plugin for Eclipse (see section 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 (http://www.delorie.com/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 (http://www.delorie.com/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 and DragonFly

FreeBSD (http://www.freebsd.org)'s ports collection (http://www.freebsd.org/ports) has an Ada framework with an expanding set of software packages. The Framework is currently built by FSF GCC 6.3.1, although FSF GCC 5.4 can optionally be used instead. The AdaCore GPL compilers are not present. There are several reasons for this, not the least of which is the addition maintenance of multiple compilers is significant. There are no non-GCC based Ada compilers represented in ports either.

While FreeBSD does have a snapshot that goes with each release, the ports are updating in a rolling fashion continuously, and the vast majority of users prefer the "head" of ports which has the latest packages.

Languages supported	Ada 83, Ada 95, Ada 2005, Ada 2012, C, C++, ObjC, Fortran
License for the run-time library	GPLv3 with Runtime Library Exception v3
Native platforms	FreeBSD i386, FreeBSD AMD64, FreeBSD ARM64, DragonFly x86-64
Cross platforms	FreeBSD/DragonFly->Android (targets ARMv7 and x86), FreeBSD/DragonFly->FreeBSD/ARM64 (targets Aarch64)
Compiler back-end	GCC 6.3.1
Available from	http://www.freebsd.org, https://github.com/DragonFlyBSD/DPorts
Support	Volunteer; public bug database

There are two ways to install the software. The quickest and easiest way is to install prebuilt binaries using the command "pkg install <pkg name>". For example, to install the GNAT Programming Studio and all of its dependencies including the GNAT compiler, all you need is one command:

```
pkg install gps-ide
```

If a specific package is not available, or the user just prefers to build from source (this can take a long time), then a typical command would be:

```
cd /usr/ports/devel/gps && make install clean
```

As with the binary installation, if any dependencies are missing they will be built first, also from source.

Available software as of 8 February 2017

Directory	Common Name	version	pkg name
archivers/zip-ada	Zip-Ada (Library)	52	zip-ada
cad/ghdl	GNU VHDL simulator	0.33	ghdl
databases/adabase	Thick bindings to Postgres, MySQL and SQLite	3.0	adabase
databases/apq	Ada95 database interface library	3.2.0	apq
databases/apq-mysql	APQ MySQL driver	3.2.0	apq-mysql
databases/apq-odbc	APQ ODBC driver	3.2.0	apq-odbc
databases/apq-pgsql	APQ PostgreSQL driver	3.2.0	apq-pgsql
devel/ada-util	Ada 2005 app utilities (Library)	1.8.0	ada-util
devel/adaid	UUID generation library	0.0.1	adaid
devel/adabooch	Ada95 Booch Components (Library)	2016-03-21	adabooch
devel/adacurses	AdaCurses (Binding)	2015-08-08	adacurses
devel/afay	AFlex and AYACC parser generators	041111	afay
devel/ahven	Ahven (Unit Test Library)	2.6	ahven
devel/alog	Stackable logging framework	0.5.2	alog
devel/aunit	Unit testing framework	2016	aunit
devel/florist-gpl	Florist (Posix Binding)	2016	florist-gpl
devel/gnatcoll	GNAT Component Collection	2016	gnatcoll
devel/gnatpython	GNATPython (python-based test framework)	2014-02-24	gnatpython
devel/gprbuild	GPRbuild (Multi-language build tool)	20160609	gprbuild
devel/gps	GNAT Programming Studio	2016	gps-ide
devel/libspark2012	SPARK 2012 library source files	2012	libspark2012
devel/matreshka	Matreshka (Info Systems Library)	0.7.0	matreshka
devel/pcsc-ada	PCSC library	0.7.3	pcsc-ada
devel/pragmarcs	PragmAda Reusable Components	20161207	pragmarcs
devel/sdl_gnat	GNAT SDL bindings (Thin)	2013	sdl_gnat
devel/simple_components	Simple Ada components	4.18	simple_components

dns/ironsides	Spark/Ada Ironsides DNS Server	2015-04-15	ironsides
graphics/generic_image_decoder	image decoder library	05	generic_image_decoder
lang/adacontrol	AdaControl (Construct detection tool)	1.17r3	adacontrol
lang/asis	Ada Semantic Interface Specification	2016	asis
lang/gcc5-aux	GNAT Ada compiler (FSF GCC)	5.4 (2016-06-03)	gcc5-aux
lang/gcc6-aux	GNAT Ada compiler (FSF GCC)	6.3.1 (2017-02-02)	gcc6-aux
lang/gnat_util	GNAT sources (helper Library)	2017-02-02	gnat_util
lang/gnatcross-aarch64	FreeBSD/ARM64 cross-compiler, Aarch64	2017-02-02 (6.3.1)	gnatcross-aarch64
lang/gnatcross-binutils-aarch64	GNU Binutils used by FreeBSD/ARM64 cross-compiler	2.27	gnatcross-binutils-aarch64
lang/gnatcross-sysroot-aarch64	FreeBSD/ARM64 sysroot	1	gnatcross-sysroot-aarch64
lang/gnatdroid-armv7	Android 5.0 cross-compiler, ARMv7	2017-02-02 (6.3.1)	gnatdroid-armv7
lang/gnatdroid-binutils	GNU Binutils used by Android cross-compiler	2.27	gnatdroid-binutils
lang/gnatdroid-binutils-x86	GNU Binutils used by Android cross-compiler (x86)	2.27	gnatdroid-binutils-x86
lang/gnatdroid-sysroot	Android API 4.0 to 6.0 sysroot	23	gnatdroid-sysroot
lang/gnatdroid-sysroot-x86	Android API 4.4 to 6.0 sysroot (x86)	23	gnatdroid-sysroot-x86
lang/gnatdroid-x86	Android 5.0 cross-compiler, x86	2017-02-02 (6.3.1)	gnatdroid-x86
lang/lua-ada	Ada bindings for Lua	1.0	ada-lua
math/plplot-ada	PLplot Ada bindings	5.12.0	plplot-ada
misc/excel-writer	Excel output library	15	excel-writer
misc/ini_file_manager	Configuration file library	03	ini_file_manager
net/adasockets	IPv4 socket library	1.10	adasockets
net/anet	Network library (IPv4 and IPv6)	0.3.4	anet
net/polyorb	PolyORB (CORBA/SOAP/DSA middleware)	2.11.1 (2014)	polyorb
security/libadacrypt	Cryptography Library (symm & asymm)	20151019	libadacrypt
security/libsparkcrypto	LibSparkCrypto (Cryptography Library)	0.1.1	libsparkcrypto
shells/sparforte	Shell and scripting language for mission-critical projects	2.0.2	spareforte

textproc/adabrowse	AdaBrowse (Ada95 HTML doc. generator)	4.0.3	adabrowse
textproc/opentoken	Ada Lex analyzer and parser	6.0b	opentoken
textproc/py-sphinxcontrib-adadomain	Sphinx documentation generator for Ada	0.1	py27-sphinxcontrib-adadomain
textproc/templates_parser	AWS Template Parser library	17.0.0	templates_parser
textproc/words	Words (Latin/English dictionary)	1.97F	words
textproc/xml_ez_out	XML output (Library)	1.06	xml_ez_out
textproc/xmlada	XML/Ada (Library)	17.0.0	xmlada
www/aws	Ada Web Server	17.0.1	aws
www/aws-demos	Ada Web Server demos	17.0.1	aws-demos
x11-toolkits/gtkada	GTK2/Ada (bindings)	2.24.4	gtkada
x11-toolkits/gtkada3	GTK3/Ada (bindings)	3.14.2	gtkada3

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/ (see other Gentoo dev-ada (http://es.znurt.org/dev-ada) packages)
Support	Volunteer; public bug database
Add-ons included	AdaBindX, AdaBroker, AdaDoc, AdaOpenGL, AdaSockets, ASIS, AUnit, Booch Components, CBind, Charles, Florist, GLADE, GPS, GtkAda, XML/Ada

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 (http://mingw.sourceforge.net) contains a version of the GNAT compiler.

The current version of MinGW (5.1.6) contains gcc-4.5.0. This includes a fully functional GNAT compiler. If the automatic downloader does not work correctly you can download the compiler directly: pick gcc-4.5.0-1 from MinGW/BaseSystem/GCC/Version4/

old instructions

The following list should help you with the installation. (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*

2. Install *mingw32-make-3.80.0-3.exe (Optional)*
3. Install *gdb-5.2.1-1.exe (Optional)*
4. Install *MSYS-1.0.10.exe (Optional)*
5. Install *msysDTK-1.0.1.exe (Optional)*

1. extract *msys-automake-1.8.2.tar.bz2 (Optional)*
2. extract *msys-autoconf-2.59.tar.bz2 (Optional)*
3. 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 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.

OpenCSW (for Solaris on SPARC and x86)

OpenCSW (http://www.opencsw.org) has binary packages of GCC 3.4.6 and 4.6.2 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	Oracle Solaris and OpenSolaris on SPARC and x86
Cross platforms	none
Compiler back-end	GCC 3.4.6 and 4.6.2 (both available)
Support	?
Available from	http://www.opencsw.org/
Add-ons included	none (?)

pkgsrc: NetBSD, DragonFly, FreeBSD and Solaris

The pkgsrc (http://www.pkgsrc.org) portable package file system has a small Ada framework. It is based on FSF GCC 5.4 currently and the FSF GCC 6.2 is available as well. The AdaCore GPL versions are not present, nor are non-GCC based compilers.

The pkgsrc system is released in quarterly branches, which are normally recommended. However, a user could also choose the "head" which would the very latest package versions. The pkgsrc system supports 21 platforms, but for Ada this is potentially limited to 5 due to the bootstrap compiler requirement: NetBSD, DragonFly, SunOS (Solaris/Illumos), OpenBSD/MirBSD, and FreeBSD.

Languages supported	Ada 83, Ada 95, Ada 2005, Ada 2012, C, C++, ObjC, Fortran
License for the run-time library	GPLv3 with Runtime Library Exception v3
Native platforms	NetBSD i386 and amd64, DragonFly x86-64, FreeBSD i386 and amd64, Solaris i386 and x86_64
Cross platforms	None
Compiler back-end	GCC 5.4 (GCC 4.9 and 6 available)
Available from	http://www.pkgsrc.org, status: http://www.pkgsrc.se
Support	Volunteer; public bug database

There are two ways to install the software. The quickest and easiest way is to install prebuilt binaries using the command "pkg_add <pkg name>". For example, to install the GNAT Programming Studio and all of its dependencies including the GNAT compiler, all you need is one command:

```
pkg_add gps
```

If a specific package is not available, or the user just prefers to build from source (this can take a long time), then a typical command would be:

```
cd /usr/pkg/devel/gps && bmake install
```

As with the binary installation, if any dependencies are missing they will be built first, also from source.

Available software as of 14 December 2016

Directory	Common Name	version	pkg name
cad/ghdl	GNU VHDL simulator	0.32rc1	ghdl
devel/florist	Florist (Posix Binding)	2012	florist-gpl
devel/gnatpython	GNATPython (python-based test framework)	2011-09-12	gnatpython
devel/gprbuild-aux	GPRbuild (Multi-language build tool)	2016-06-09	gprbuild-aux
lang/gcc-aux	GNAT Ada compiler (FSF GCC)	4.9.2 (2014-10-23)	gcc-aux
lang/gcc5-aux	GNAT Ada compiler (FSF GCC)	5.4.0 (2016-06-03)	gcc5-aux
lang/gcc6-aux	GNAT Ada compiler (FSF GCC)	6.2.0 (2016-08-22)	gcc6-aux
textproc/xmlada	XML/Ada (Library)	4.4.0	xmlada
www/aws	Ada Web Server	3.1.0.0 (w)	aws
www/aws-demos	Ada Web Server demos	3.1.0.0 (w)	aws-demos
x11/gtkada	GTK/Ada (bindings)	2.24.4	gtkada

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 SuSE version 12.1, the compiler is in the package

```
gcc46-ada
libada46
```

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

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

Ubuntu

Ubuntu (and derivatives like Kubuntu, Xubuntu...) is a Debian-based Linux distribution, thus the installation process described above can be used. Graphical package managers like Synaptic or Adept can also be employed to select the Ada packages.

ICC from Irvine Compiler Corporation

Irvine Compiler Corporation (http://www.irvine.com/) provides native and cross compilers for various platforms.[5] (http://www.irvine.com/products.html) The compiler and run-time system support development of certified, safety-critical software.

Commercial, proprietary. No-cost evaluation is possible on request. Royalty-free redistribution of the run-time system is allowed.

Janus/Ada 83 and 95 from RR Software

RR Software (http://www.rrsoftware.com) 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.

Janus/Ada 95 supports subset of Ada 2007 and Ada 2012 features (http://www.rrsoftware.com/html/blog/ja-321-rel.html).

Commercial but relatively cheap; proprietary.

MAXAda from Concurrent

Concurrent (https://www.concurrent-rt.com) offers MAXAda (https://www.concurrent-rt.com/wp-content/uploads/2016/11/maxada-data-sheet.pdf), an Ada 95 compiler for Linux/Xeon and PowerPC platforms, and Ada bindings to POSIX and X/Motif.[6] (https://redhawk.concurrent-rt.com/docs/root/1Linux/5Compilers/Ada/0898539-3.5.1-SR3.pdf)

Commercial, proprietary.

ObjectAda from PTC (formerly Aonix/Atego)

PTC offers ObjectAda (https://www.ptc.com/en/products/developer-tools/objectada) native (Windows, some flavors of Unix, and Linux) and cross (PPC, Intel, VxWorks, and ERC32) compilers.

Limited support of Ada 2012 is available.

Commercial, proprietary.

PowerAda from OC Systems

OC Systems (http://www.ocsystems.com/) offers Ada compilers and bindings to POSIX and X-11:

- PowerAda (http://www.ocsystems.com/prod_powerada.html), an Ada 95 compiler for Linux and AIX,
- LegacyAda/390 (http://www.ocsystems.com/prod_legacyada.html), an Ada 83 compiler for IBM System 370 and 390 mainframes

Commercial, proprietary.

ApexAda from PTC (formerly IBM Rational)

PTC ApexAda (https://www.ptc.com/en/products/developer-tools/apexada) for native and embedded development.

Commercial, proprietary.

SCORE from DDC-I

DDC-I (http://www.ddci.com/) offers its SCORE (https://www.ddci.com/products_score) cross-compilers for embedded development. SCORE stands for Safety-Critical, Object-oriented, Real-time Embedded.

Commercial, proprietary.

TADS from Tartan

Tartan (http://tartan.com/index.html) offers the Tartan Ada Development System (TADS), with cross-compilers for some digital signal processors.

Commercial, proprietary.

XD Ada from DXC (http://www.dxc.technology)

XD Ada (http://assets1.dxc.technology/manufacturing/downloads/MD_6933a-18_XD_Ada_Cross-Compiler_OO_final.pdf) is an Ada 83 cross-compiler for embedded development. Hosts include VAX, Alpha and Integrity Servers running OpenVMS. Targets include Motorola 68000 and MIL-STD-1750A processors.

Commercial, proprietary.

XGC Ada from XGC Software

XGC compilers are GCC with custom run-time libraries suitable for avionics and space applications. The run-time kernels are very small and do not support exception propagation (i.e. you can handle an exception only in the subprogram that raised it).

Commercial but some versions are also offered as free downloads. Free Software.

Languages supported	Ada 83, Ada 95, C
License for the run-time library	GNAT-Modified GPL
Native platforms	none
Cross platforms	Hosts: sun-sparc-solaris, pc-linux2.*; targets are bare boards with ERC32, MIL-STD-1750A, Motorola 68000 family or Intel 32-bit processors. PowerPC and Intel 80186 targets on request.
Compiler back-end	GCC 2.8.1
Available from	http://www.xgc.com/
Support	Commercial
Add-ons included	Ravenscar-compliant run-time kernels, certified for avionics and space applications; gdb cross-debugger; target simulator.

References

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

> *This list is incomplete. You can help Wikibooks by adding the build information (https://en.wikibooks.org/w/index.php?title=Ada_Programming/Building&action=edit) for other compilers.*

Alire

Alire (https://alire.ada.dev/) is a build tool and package manager which automatically manages package dependencies including downloading the needed build tools. It is based on the GNAT compiler.

You can create a new project with `alr init`:

Code:

```
alr init test
```

Output:

```
Select the kind of crate you want to create:
  1. LIBRARY
  2. BINARY
Enter your choice index (first is default):
> 1
Enter a short description of the crate: (default: '')
> Testing alr init
Select a software license for the crate?
  1. MIT OR Apache-2.0 WITH LLVM-exception
  2. MIT
  3. Apache-2.0 WITH LLVM-exception
  4. Apache-2.0
  5. BSD-3-Clause
  6. LGPL-3.0-or-later
  7. GPL-3.0-or-later WITH GCC-exception-3.1
  8. GPL-3.0-or-later
  9. Other...
Enter your choice index (first is default):
> 8
Enter a comma (',') separated list of tags to help people find your crate: (default: '')
> test
Enter an optional Website URL for the crate: (default: '')
> https://wikibook-ada.sourceforge.net/
✓ test initialized successfully.
```

Alire supports most of the GNAT feature mentioned below as well as the same IDE and code editors.

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 -c -f -u
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 called GPS, included with recent releases of the GPL version of GNAT. GPS features a graphical user interface.

Emacs includes (Ada Mode (http://stephe-leake.org/emacs/ada-mode/emacs-ada-mode.html)), and GNAT plugins for KDevelop and Vim (Ada Mode (http://www.vim.org/scripts/script.php?script_id=1609)) are available.

Vim Ada Mode is maintained by The GNU Ada project (http://gnuada.sourceforge.net).

GNAT with Xcode

Apple's free (gratis) IDE, Xcode, uses the LLVM compiler with the Clang front-end, and does not support Ada: however, in Xcode 4.3 for OS X Lion and later versions, the command line tools (assembler, linker etc) which are required to use GNAT are an optional component of Xcode and must be specially installed (https://developer.apple.com/library/ios/documentation/DeveloperTools/Conceptual/WhatsNewXcode/Articles/xcode_4_3.html).

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 (http://h71000.www7.hp.com/commercial/ada/ada_index.html)”. 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 syntactically correct enough for the compiler to determine the packages declared and analyze the `with` statements. Dec Ada allows you to have more than 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 but 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.

DEC Ada IDE

Dec Ada comes without an IDE, however the DEC LSE as well as the Ada Mode (http://www.vim.org/scripts/script.php?script_id=1609) of the Vim text editor support DEC Ada.

Compiling our Demo Source

Once you have downloaded (https://sourceforge.net/project/showfiles.php?group_id=124904) our example programs, you might wonder how to compile them.

Unless you use Alire (https://alire.ada.dev/), 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 IDE not named, it might be in the making and not actually work).

Alire

Alire (https://alire.ada.dev/) is the simplest way to compile the sample code if you use Windows, macOS or Linux. With Alire, you don't even need to download and extract the source, as Alire will do everything for you. All you need to do after the installation of Alire itself (https://alire.ada.dev/docs/#installation) is execute the following command in a terminal or CMD window:

Example for Unix like operating systems

Code:

```
alr get "wikibook"
```

Output:

```
ⓘ Deploying wikibook=1.0.1...
########################################################################################################################################################################
######################
100.0%#################################################################################################################################################################
############################ 100.0%

wikibook=1.0.1 successfully retrieved.
ⓘ Found 2 nested crates in /Work/wikibook_1.0.1_780ee70f:
   basic/basic=1.0.1: Samples for WikiBook Ada Programing: Basic Ada
   pragmas_restrictions/pragmas_restrictions=1.0.1: Samples for WikiBook Ada Programing: Pragmas Restrictions
Dependencies were solved as follows:

   + gnat 13.2.1 (new,gnat_native,binary)
```

This will download all samples ordered by chapter. To then, for example, build and execute the first sample code from the Basic Ada chapter you type:

Example for Unix like operating systems

Code:

```
cd "wikibook_1.0.1_780ee70f/basic"
alr build
bin/hello_word_1
```

Output:

```
ⓘ Synchronizing workspace...
Dependencies automatically updated as follows:

   + gnat 13.2.1 (new,gnat_native,binary)

ⓘ Building basic=1.0.1/basic.gpr...
Setup
   [mkdir]        object directory for project Basic
   [mkdir]        exec directory for project Basic
Compile
   [Ada]          hello_world_1.adb
   [Ada]          hello_world_2.adb
   [Ada]          hello_world_3.adb
```

```
Bind
   [gprbind]      hello_world_1.bexch
   [gprbind]      hello_world_2.bexch
   [gprbind]      hello_world_3.bexch
   [Ada]          hello_world_1.ali
   [Ada]          hello_world_2.ali
   [Ada]          hello_world_3.ali
Link
   [link]         hello_world_1.adb
   [link]         hello_world_2.adb
   [link]         hello_world_3.adb
✓ Build finished successfully in 0.90 seconds.

Hello World!
```

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 its 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 command-line

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 (https://sourceforge.net/p/wikibook-ada/code/HEAD/tree/trunk/demos/Source/hello_world_1.adb), plain text (https://sourceforge.net/p/wikibook-ada/code/HEAD/tree/trunk/demos/Source/hello_world_1.adb?format=raw), download page (https://sourceforge.net/project/showfiles.php?group_id=124904), browse all (https://wikibook-ada.sourceforge.net/samples))

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):

- 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.

- Compile the source file:

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

- 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.

External links

- GNAT Online Documentation:
  - GNAT User's Guide (http://gcc.gnu.org/onlinedocs/gcc-4.0.1/gnat_ugn_unw/)
- DEC Ada:
  - Developing Ada Products on OpenVMS (http://h71000.www7.hp.com/commercial/ada/ada_dap.pdf) (PDF)
  - DEC Ada — Language Reference Manual (http://h71000.www7.hp.com/commercial/ada/ada_lrm.pdf) (PDF)
  - DEC Ada — Run-Time Reference (http://h71000.www7.hp.com/commercial/ada/ada_rtr.pdf) (PDF)

Control Statements

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
    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 ("Wow!");
    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, without checking any other condition . 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;
```

The subtype of X must be a discrete type, i.e. an enumeration or integer type.

In Ada, one advantage of the case statement is that the compiler will check the coverage of the choices, that is, all the values of the subtype of variable X must be present or a default branch `when others` must specify what to do in the remaining cases.

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 control to the statement after the label.

```
   goto Label;

   Dont_Do_Something;

<<Label>>
   ...
```

Is *goto* evil?

Since Professor Dikstra wrote his article Go To Statement Considered Harmful (https://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf), *goto* is considered bad practice in structured programming languages, and, in fact, many programming style guides forbid the use of the *goto* statement. 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. There is an important difference though: a return is a forward only use of *goto*. Exceptions are also a type of *goto* statement; and note that they need not specify where they are going to.

Therefore, if you have functions and procedures with more than one *return* statement you are breaking the structure of the program similarly to a use of *goto*. In practice, nearly every programmer is familiar with a 'return' statement and its associated behavior; thus, when it comes down to readability the following two samples are almost the same:

Note also that the *goto* statement in Ada is safer than in other languages, since it doesn't allow you to transfer control:

- outside a body;
- between alternatives of a *case* statement, *if* statement, or *select* statement;
- between different exception handlers; or from an exception handler of a handled_sequence_of_statements back to its sequence_of_statements.

```
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. The term, *endless loop*, is a relative term; if the running program is forcibly terminated by some means beyond the control of the program, then an endless loop will indeed end.

```
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 are at 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 than 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 subtype and range as seen in the example. If you leave out the subtype then the compiler will determine it by context; if you leave out the range then the loop will iterate over every value of the subtype 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.

The loop counter I is a constant implicitly declared and ceases to exist after the body of the loop.

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 example

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

File: range_1.adb (view (https://sourceforge.net/p/wikibook-ada/code/HEAD/tree/trunk/demos/Source/range_1.adb), plain text (https://sourceforge.net/p/wikibook-ada/code/HEAD/tree/trunk/demos/Source/range_1.adb?format=raw), download page (https://sourceforge.net/project/showfiles.php?group_id=124904), browse all (https://wikibook-ada.sourceforge.net/samples))

```
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 (http://www.ada-auth.org/standards/12rm/html/RM-5-3.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-5-3.html)]
- 5.4: Case Statements (http://www.ada-auth.org/standards/12rm/html/RM-5-4.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-5-4.html)]
- 5.5: Loop Statements (http://www.ada-auth.org/standards/12rm/html/RM-5-5.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-5-5.html)]
- 5.6: Block Statements (http://www.ada-auth.org/standards/12rm/html/RM-5-6.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-5-6.html)]
- 5.7: Exit Statements (http://www.ada-auth.org/standards/12rm/html/RM-5-7.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-5-7.html)]
- 5.8: Goto Statements (http://www.ada-auth.org/standards/12rm/html/RM-5-8.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-5-8.html)]
- 6.5: Return Statements (http://www.ada-auth.org/standards/12rm/html/RM-6-5.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-6-5.html)]

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:

- **Type**: a way to categorize data. characters are types 'a' through 'z'. Integers are types that include 0,1,2....
- **Strong typing**: types are incompatible with one another, so it is not possible to mix apples and oranges. The compiler will not guess that your apple is an orange. You must explicitly say my_fruit = fruit(my_apple). Strong typing reduces the amount of errors. This is because developers can really easily write a float into an integer variable without knowing. Now data you needed for your program to succeed has been lost in the conversion when the complier switched types. Ada gets mad and rejects the developer's dumb mistake by refusing to do the conversion unless explicitly told.
- **Static typing**: type checked while compiling, this allows type errors to be found earlier.
- **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. An example of an abstraction is your car. You don't really know how it works you just know that bumbling hunk of metal moves. Nearly every technology you work with is abstracted layer to simplify the complex circuits that make it up - the same goes for software. You want abstraction because code in a class makes a lot more sense than a hundred if statements with no explanation when debugging
- **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 exactly the 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 their base type and may be compatible with one another. See below for examples of subtypes which are incompatible with one another.

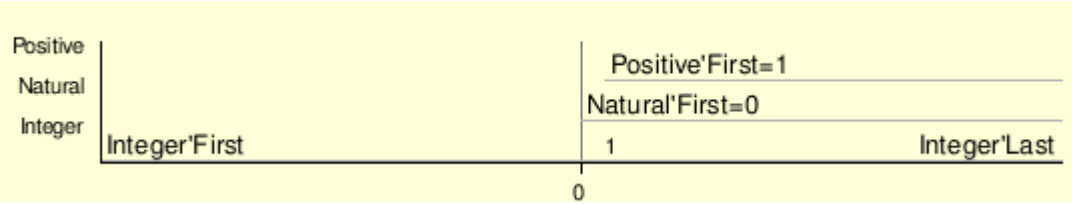
Predefined types

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) (http://www.ada-auth.org/standards/12rm/html/RM-3-5-4.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-3-5-4.html)]). 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) (http://www.ada-auth.org/standards/12rm/html/RM-3-5-7.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-3-5-7.html)]); 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) (http://www.ada-auth.org/standards/12rm/html/RM-A-1.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-A-1.html)]).

Character
: A special form of Enumerations. There are three predefined kinds of character types: 8-bit characters (called `Character`), 16-bit characters (called `Wide_Character`), and 32-bit characters (`Wide_Wide_Character`). `Character` has been present since the first version of the language (Ada 83), `Wide_Character` was added in Ada 95, while the type `Wide_Wide_Character` is available with Ada 2005.

String
: Three indefinite array types, of `Character`, `Wide_Character`, and `Wide_Wide_Character` respectively. 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 subtract one address from another to get the offset between them. Together, `Address`, `Storage_Offset` and their associated subprograms provide for address arithmetic.

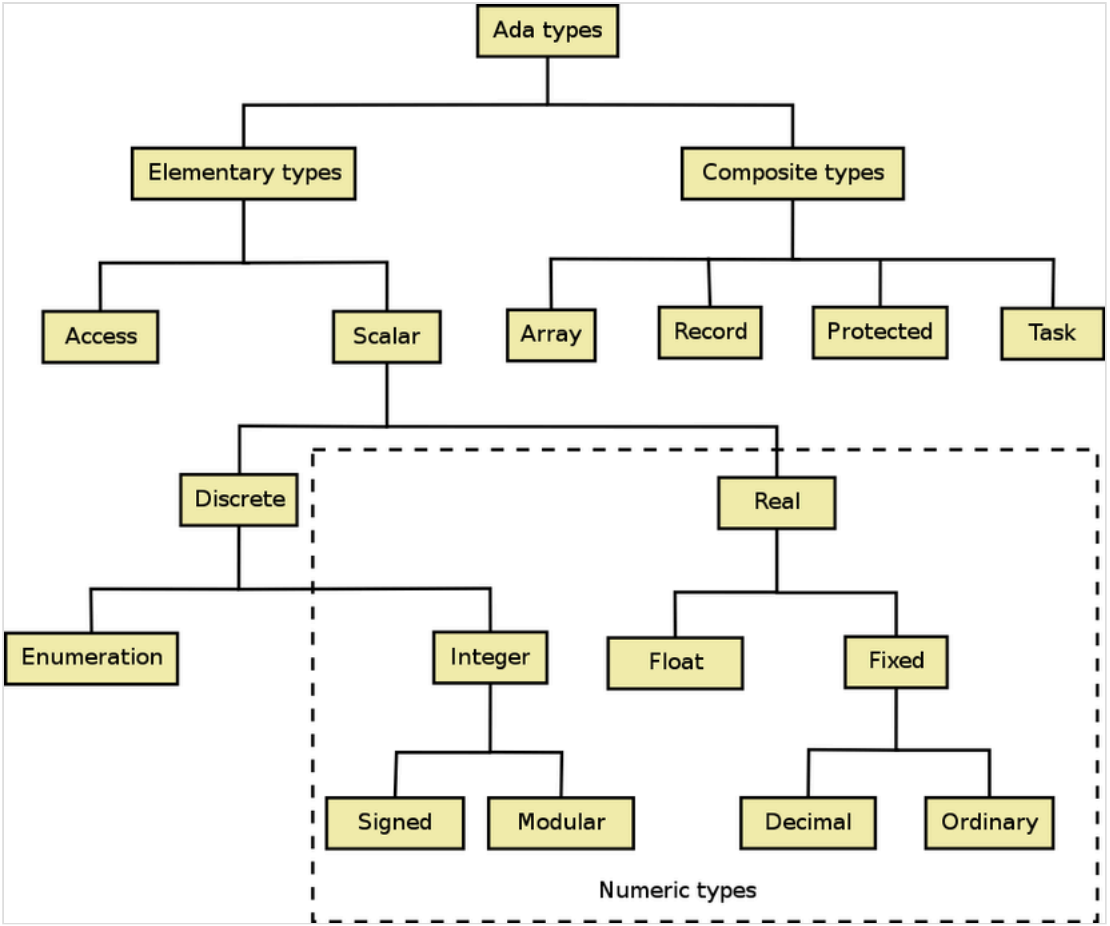
System.Storage_Elements.Storage_Count
: A subtype of `Storage_Offset` which cannot be negative, and represents the memory size of a data structure (similar to C's `size_t`).

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 (similar to multithreading in C++)
: Task and Protected types allow the control of concurrency

Interfaces (similar to virtual methods in C++)
: New in Ada 2005, these types are similar to the Java interfaces.

Classification of Types

Ada's types can be classified as follows.

Specific vs. Class-wide

```
type T is ...  --  a specific type
  T'Class      --  the corresponding class-wide type (exists only for tagged types)
```

`T'Class` and `T'Class'Class` are the same.

Primitive operations with parameters of specific types are *non-dispatching*, those with parameters of class-wide types are *dispatching*.

New types can be declared by *deriving* from specific types; primitive operations are *inherited* by derivation. You cannot derive from class-wide types.

Constrained vs. Unconstrained

```
type I is range 1 .. 10;           --  constrained
type AC is array (1 .. 10) of ...  --  constrained
```

```
type AU is array (I range <>) of ...          --  unconstrained
type R (X: Discriminant [:= Default]) is ...  --  unconstrained
```

By giving a *constraint* to an unconstrained subtype, a subtype or object becomes constrained:

```
subtype RC is R (Value);  --  constrained subtype of R
OC: R (Value);            --  constrained object of anonymous constrained subtype of R
OU: R;                    --  unconstrained object
```

Declaring an unconstrained object is only possible if a default value is given in the type declaration above. The language does not specify how such objects are allocated. GNAT allocates the maximum size, so that size changes that might occur with discriminant changes present no problem. Another possibility is implicit dynamic allocation on the heap and re-allocation followed by a deallocation when the size changes.

Definite vs. Indefinite

```
type I is range 1 .. 10;                    --  definite
type RD (X: Discriminant := Default) is ...  --  definite
```

```
type T (<>) is ...                    --  indefinite
type AU is array (I range <>) of ...  --  indefinite
type RI (X: Discriminant) is ...      --  indefinite
```

Definite subtypes allow the declaration of objects without initial value, since objects of definite subtypes have constraints that are known at creation-time. Object declarations of indefinite subtypes need an initial value to supply a constraint; they are then constrained by the constraint delivered by the initial value.

```
OT: T  := Expr;                       --  some initial expression (object, function call, etc.)
OA: AU := (3 => 10, 5 => 2, 4 => 4);  --  index range is now 3 .. 5
OR: RI := Expr;                       --  again some initial expression as above
```

Unconstrained vs. Indefinite

Note that unconstrained subtypes are not necessarily indefinite as can be seen above with RD: it is a definite unconstrained subtype.

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, correctly called the "type of T", is anonymous; the RM refers to it as *T* (in italics), but often speaks sloppily about the type T. But this is an academic consideration; for most purposes, it is sufficient to think of `T` as a type. For scalar types, there is also a base type called `T'Base`, which encompasses all values of T.

For signed integer types, the type of T comprises the (complete) set of mathematical integers. The base type is a certain hardware type, symmetric around zero (except for possibly one extra negative value), encompassing all values of T.

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;  -- bad
subtype Integer_3 is Integer_1'Base range 7 .. 11;  -- OK
A : Integer_1 := 8;
B : Integer_3 := A; -- OK
```

The declaration of `Integer_2` is bad because the constraint `7 .. 11` is not compatible with `Integer_1`; it raises `Constraint_Error` at subtype elaboration time.

`Integer_1` and `Integer_3` are compatible because they are both subtypes of the same type, namely `Integer_1'Base`.

It is not necessary that the subtype 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_3`, 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
      -- procedure P (I: in Integer_2);  inherited P implicitly defined here
   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)`, the parameter B is converted to `Integer_1`; this conversion of course passes since the set of acceptable values for the derived type (here, 8 .. 10) must be included in that of the parent type (1 .. 10). Then P is called with the converted parameter.

Consider however a variant of the example above:

```
procedure Derived_Types is

  package Pak is
    type Integer_1 is range 1 .. 10;
    procedure P (I: in Integer_1; J: out Integer_1);
    type Integer_2 is new Integer_1 range 8 .. 10;
  end Pak;
```

```
   package body Pak is
     procedure P (I: in Integer_1; J: out Integer_1) is
     begin
       J := I - 1;
     end P;
   end Pak;

   use Pak;

   A: Integer_1 := 4;  X: Integer_1;
   B: Integer_2 := 8;  Y: Integer_2;

begin

   P (A, X);
   P (B, Y);

end Derived_Types;
```

When `P (B, Y)` is called, both parameters are converted to `Integer_1`. Thus the range check on J (7) in the body of P will pass. However on return parameter Y is converted back to `Integer_2` and the range check on Y will of course fail.

With the above in mind, you will see why in the following program Constraint_Error will be called at run time, before `P` is even called.

```
procedure Derived_Types is

   package Pak is
     type Integer_1 is range 1 .. 10;
     procedure P (I: in Integer_1; J: out Integer_1);
     type Integer_2 is new Integer_1'Base range 8 .. 12;
   end Pak;

   package body Pak is
     procedure P (I: in Integer_1; J: out Integer_1) is
     begin
       J := I - 1;
     end P;
   end Pak;

   use Pak;

   B: Integer_2 := 11;  Y: Integer_2;

begin

   P (B, Y);
```

```
end Derived_Types;
```

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

In Ada, all types are anonymous and only subtypes may be named. For scalar types, there is a special subtype of the anonymous type, called the *base type*, which is nameable with `Subtype'Base` notation. This `Name'Attribute` (read "name tick attribute") is the special notation used in Ada for what is called an attribute, i.e. a characteristic of a type, a variable, or some other program entity, that is defined by the compiler and can be queried. In this case, the base type (`Subtype'Base`) comprises all values of the *first subtype*. Some examples:

```
  type Int is range 0 .. 100;
```

The base type `Int'Base` is a hardware type selected by the compiler that comprises the values of `Int`. Thus, it may have the range -2^7 .. 2^7-1 or -2^{15} .. $2^{15}-1$ or any other such type.

```
  type Enum  is (A, B, C, D);
  type Short is new Enum range A .. C;
```

`Enum'Base` is the same as `Enum`, but `Short'Base` also holds the literal `D`.

Constrained subtype

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

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

You cannot partially constrain an unconstrained subtype:

```
type My_Array is array (Integer range <>, Integer range <>) of Some_Type;

--  subtype Constr is My_Array (1 .. 10, Integer range <>);  illegal

subtype Constr is My_Array (1 .. 10, -100 .. 200);
```

Constraints for all indices must be given, the result is necessarily 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`.

A subtype of an indefinite subtype that does not add a constraint only introduces a new name for the original subtype (a kind of renaming under a different notion).

```
  subtype My_String is String;
```

`My_String` and `String` are interchangeable.

Named subtype

A subtype which has a name assigned to it. “First 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), others with the keyword **`subtype`**. For example:

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

`Count_to_Ten` is the first subtype of a suitable integer base type. However, if you 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 its 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

Any indefinite type is also an unconstrained subtype. However, unconstrainedness and indefiniteness are not the same.

```
type My_Enum is (A, B, C);
type My_Record (Discriminant: My_Enum) is ...;

My_Object_A: My_Record (A);
```

This type is unconstrained and indefinite because you need to give an actual discriminant for object declarations; the object is constrained to this discriminant which may not change.

When however a default is provided for the discriminant, the type is definite yet unconstrained; it allows to define both, constrained and unconstrained objects:

```
type My_Enum is (A, B, C);
type My_Record (Discriminant: My_Enum := A) is ...;

My_Object_U: My_Record;      --  unconstrained object
My_Object_B: My_Record (B);  --  constrained to discriminant B like above
```

Here, My_Object_U is unconstrained; upon declaration, it has the discriminant A (the default) which however may change.

Incompatible subtypes

```
type My_Integer is range -10 .. + 10;
subtype My_Positive is My_Integer range + 1 .. + 10;
subtype My_Negative is My_Integer range -10 .. -  1;
```

These subtypes are of course incompatible.

Another example are subtypes of a discriminated record:

```
type My_Enum is (A, B, C);
type My_Record (Discriminant: My_Enum) is ...;
subtype My_A_Record is My_Record (A);
subtype My_C_Record is My_Record (C);
```

Also these subtypes are incompatible.

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 is 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 (https://sourceforge.net/p/wikibook-ada/code/HEAD/tree/trunk/demos/Source/convert_evaluate_as.adb), plain text (https://sourceforge.net/p/wikibook-ada/code/HEAD/tree/trunk/demos/Source/convert_evaluate_as.adb?format=raw), download page (https://sourceforge.net/project/showfiles.php?group_id=124904), browse all (https://wikibook-ada.sourceforge.net/samples))

```
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;
```

You should use qualified expression when getting a string literal's length.

```
"foo"'Length                 {{Ada/--| compilation error: prefix of attribute must be a name}}
                             {{Ada/--|                    qualify expression to turn it into a name}}
String'("foo" & "bar")'Length {{Ada/--| 6}}
```

Type conversions

Data do not always come in the format you need them. You must, then, face the task of converting them. As a true multi-purpose language with a special emphasis on "mission critical", "system programming" and "safety", Ada has several conversion techniques. The most difficult part is choosing the right one, so the following list is sorted in order of utility. You should try the first one first; the last technique is a last resort, to be used if all others fail. There are also a few related techniques that 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 examples also 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 (note special rules for tagged types)
- between array types under certain conditions (see RM 4.6(24.2/2..24.7/2))
- and *nowhere else*

(The rules become more complex with class-wide and anonymous access types.)

```
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 (https://sourceforge.net/p/wikibook-ada/code/HEAD/tree/trunk/demos/Source/convert_checked.adb), plain text (https://sourceforge.net/p/wikibook-ada/code/HEAD/tree/trunk/demos/Source/convert_checked.adb?format=raw), download page (https://sourceforge.net/project/showfiles.php?group_id=124904), browse all (https://wikibook-ada.sourceforge.net/samples))

```
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);  --  range check will lead to Constraint_Error
```

```
   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 (thus the code above will raise Constraint_Error), 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;
```

You might ask why you should convert between two subtypes of the same type. An example will illustrate this.

```
subtype String_10 is String (1 .. 10);
X: String := "A line long enough to make the example valid";
Slice: constant String := String_10 (X (11 .. 20));
```

Here, `Slice` has bounds 1 and 10, whereas `X (11 .. 20)` has bounds 11 and 20.

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 in the direction to the root are possible, which of course cannot fail. There are no conversions in the opposite direction (where would you get the further components from?); *extension aggregates* have to be used instead.

With the conversion itself, no components of the source object that are not present in the target object are lost, they are just hidden from visibility. Therefore, this kind of conversion is called a *view conversion* since it provides a view of the source object as an object of the target type (especially it does not change the object's tag).

It is a common idiom in object oriented programming to rename the result of a view conversion. (A renaming declaration does not create a new object; it only gives a new name to something that already exists.)

```
type Parent_Type is tagged record
   <components>;
end record;
type Child_Type is new Parent_Type with record
   <further components>;
end record;

Child_Instance : Child_Type;
Parent_View    : Parent_Type renames Parent_Type (Child_Instance);
Parent_Part    : Parent_Type := Parent_Type (Child_Instance);
```

`Parent_View` is not a new object, but another name for `Child_Instance` viewed as the parent, i.e. only the parent components are visible, the child-specific components are hidden. `Parent_Part`, however, is an object of the parent type, which of course has no storage for the child-specific components, so they are lost with the assignment.

All types derived from a tagged type `T` form a tree rooted at `T`. The class-wide type `T'Class` can hold any object within this tree. With class-wide types, conversions in any direction are possible; there is a run-time tag check that raises `Constraint_Error` if the check fails. These conversions are also view conversions, no data is created or lost.

```
Object_1 : Parent_Type'Class := Parent_Type'Class (Child_Instance);
Object_2 : Parent_Type'Class renames Parent_Type'Class (Child_Instance);
```

`Object_1` is a new object, a copy; `Object_2` is just a new name. Both objects are of the class-wide type. Conversions to any type within the given class are legal, but are tag-checked.

```
Success : Child_Type := Child_Type (Parent_Type'Class (Parent_View));
Failure : Child_Type := Child_Type (Parent_Type'Class (Parent_Part));
```

The first conversion passes the tag check and both objects `Child_Instance` and `Success` are equal. The second conversion fails the tag check. (Conversion assignments of this kind will rarely be used; dispatching will do this automatically, see object oriented programming.)

You can perform these checks yourself with membership tests:

```
if Parent_View in Child_Type then ...
if Parent_View in Child_Type'Class then ...
```

There is also the package `Ada.Tags`.

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 source data and reinterpret them under the target type without any checks. It is **your** chore to make sure that the requirements on unchecked conversion as stated in RM 13.9 (http://www.adaic.com/standards/05rm/html/RM-13-9.html) (Annotated (http://www.adaic.com/standards/05aarm/html/AA-13-9.html)) are fulfilled; if not, the result is implementation dependent and may even lead to abnormal data. Use the 'Valid attribute after the conversion to check the validity of the data in problematic cases.

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 will output `A = -1, B = 255`. No error will be reported, but is this the result you expect?

File: convert_unchecked.adb (view (https://sourceforge.net/p/wikibook-ada/code/HEAD/tree/trunk/demos/Source/convert_unchecked.adb), plain text (https://sourceforge.net/p/wikibook-ada/code/HEAD/tree/trunk/demos/Source/convert_unchecked.adb?format=raw), download page (https://sourceforge.net/project/showfiles.php?group_id=124904), browse all (https://wikibook-ada.sourceforge.net/samples))

```
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;
```

There is of course a range check in the assignment `B := Convert (A);`. Thus if `B` were defined as `B: Byte range 0 .. 10;`, `Constraint_Error` would be raised.

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 (http://www.adaic.com/standards/05rm/html/RM-B-1.html) (Annotated (http://www.adaic.com/standards/05aarm/html/AA-B-1.html)).

The example below does the same as the example from "Unchecked Conversion".

File: convert_address_mapping.adb (view (https://sourceforge.net/p/wikibook-ada/code/HEAD/tree/trunk/demos/Source/convert_address_mapping.adb), plain text (https://sourceforge.net/p/wikibook-ada/code/HEAD/tree/trunk/demos/Source/convert_address_mapping.adb?format=raw), download page (https://sour

ceforge.net/project/showfiles.php?group_id=124904), browse all (https://wikibook-ada.sourceforge.net/samples))

```
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.

Elaborated Discussion of Types for Signed Integer Types

As explained before, a type declaration

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

declares an anonymous type *T* and its first subtype `T` (please note the italicization). *T* encompasses the complete set of mathematical integers. Static expressions and named numbers make use of this fact.

All numeric integer literals are of type `Universal_Integer`. They are converted to the appropriate specific type where needed. `Universal_Integer` itself has no operators.

Some examples with static named numbers:

```
S1: constant := Integer'Last + Integer'Last;       -- "+" of Integer
S2: constant := Long_Integer'Last + 1;             -- "+" of Long_Integer
S3: constant := S1 + S2;                           -- "+" of root_integer
S4: constant := Integer'Last + Long_Integer'Last;  -- illegal
```

Static expressions are evaluated at compile-time on the appropriate types with no overflow checks, i.e. mathematically exact (only limited by computer store). The result is then implicitly converted to `Universal_Integer`.

The literal `1` in `S2` is of type `Universal_Integer` and implicitly converted to `Long_Integer`.

`S3` implicitly converts the summands to `root_integer`, performs the calculation and converts back to `Universal_Integer`.

`S4` is illegal because it mixes two different types. You can however write this as

```
S5: constant := Integer'Pos (Integer'Last) + Long_Integer'Pos (Long_Integer'Last);  -- "+" of root_integer
```

where the Pos attributes convert the values to `Universal_Integer`, which are then further implicitly converted to `root_integer`, added and the result converted back to `Universal_Integer`.

`root_integer` is the anonymous greatest integer type representable by the hardware. It has the range `System.Min_Integer .. System.Max_Integer`. All integer types are rooted at `root_integer`, i.e. derived from it. `Universal_Integer` can be viewed as `root_integer'Class`.

During run-time, computations of course are performed with range checks and overflow checks on the appropriate subtype. Intermediate results may however exceed the range limits. Thus with `I, J, K` of the subtype `T` above, the following code will return the correct result:

```
I := 10;
J :=  8;
```

```
K := (I + J) - 12;
-- I := I + J;  --  range check would fail, leading to Constraint_Error
```

Real literals are of type `Universal_Real`, and similar rules as the ones above apply accordingly.

Relations between types

Types can be made from other types. Array types, for example, are made from two types, one for the arrays' index and one for the arrays' components. An array, then, expresses an association, namely that between one value of the index type and a value of the component type.

```
type Color is (Red, Green, Blue);
type Intensity is range 0 .. 255;

type Colored_Point is array (Color) of Intensity;
```

The type `Color` is the index type and the type `Intensity` is the component type of the array type `Colored_Point`. See array.

See also

Wikibook

- Ada Programming

Ada Reference Manual

- 3.2.1: Type Declarations (http://www.ada-auth.org/standards/12rm/html/RM-3-2-1.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-3-2-1.html)]
- 3.3: Objects and Named Numbers (http://www.ada-auth.org/standards/12rm/html/RM-3-3.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-3-3.html)]
- 3.7: Discriminants (http://www.ada-auth.org/standards/12rm/html/RM-3-7.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-3-7.html)]
- 3.10: Access Types (http://www.ada-auth.org/standards/12rm/html/RM-3-10.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-3-10.html)]

- 4.9: Static Expressions and Static Subtypes (http://www.ada-auth.org/standards/12rm/html/RM-4-9.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-4-9.html)]
- 13.9: Unchecked Type Conversions (http://www.ada-auth.org/standards/12rm/html/RM-13-9.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-13-9.html)]
- 13.3: Operational and Representation Attributes (http://www.ada-auth.org/standards/12rm/html/RM-13-3.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-13-3.html)]
- Annex K: (informative) Language-Defined Attributes (http://www.ada-auth.org/standards/12rm/html/RM-K.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-K.html)]

Integer types

A `range` is a signed integer value which ranges from a First to a last Last. It is defined as

```
range First .. Last
```

When a value is assigned to an object with such a range constraint, the value is checked for validity and Constraint_Error exception is raised when the value is not within First to Last.

When declaring a range type, the corresponding mathematical operators are implicitly declared by the language at the same place.

The compiler is free to choose a suitable underlaying hardware type for this user defined type.

Working example

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

File: range_1.adb (view (https://sourceforge.net/p/wikibook-ada/code/HEAD/tree/trunk/demos/Source/range_1.adb), plain text (https://sourceforge.net/p/wikibook-ada/code/HEAD/tree/trunk/demos/Source/range_1.adb?format=raw), download page (https://sourceforge.net/project/showfiles.php?group_id=124904), browse all (https://wikibook-ada.sourceforge.net/samples))

```
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 (http://www.adaic.com/standards/05rm/html/RM-4-4.html) (Annotated (http://www.adaic.com/standards/05aarm/html/AA-4-4.html))

- 3.5.4 Integer Types (http://www.adaic.com/standards/05rm/html/RM-3-5-4.html) (Annotated (http://www.adaic.com/standards/05aarm/html/AA-3-5-4.html))

Unsigned integer types

Description

Unsigned integers in Ada have a value range from 0 to some positive number (not necessarily 1 subtracted from some 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 (see below).

The predefined package Interfaces (RM B.2 (http://www.ada-auth.org/standards/12rm/html/RM-B-2.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-B-2.html)]) 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.

Bitwise Operations

Be very careful with bitwise operators **and**, **or**, **xor**, **not**, when the modulus is not a power of two. An example might exemplify the problem.

```
type Unsigned is mod 2**5;   -- modulus 32
X: Unsigned := 2#10110#;     -- 22
not X        = 2#01001#      -- bit reversal: 9 ( = 31 - 22 ) as expected
```

The other operators work similarly.

Now take a modulus that is not a power of two. Naive expectations about the results may lead out of the value range. As an example take again the **not** operator (see the RM for the others):

```
type Unsigned is mod 5;
X: Unsigned := 2#001#;  -- 1, bit reversal: 2#110# = 6 leads out of range
```

The definition of **not** is therefore:

```
 not X = Unsigned'Last - X  -- here: 4 - 1 = 2#011#
```

See also

Wikibook

- Ada Programming
- Ada Programming/Types

- Ada Programming/Keywords/mod

Ada Reference Manual

- 4.4: Expressions (http://www.ada-auth.org/standards/12rm/html/RM-4-4.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-4-4.html)]
- 3.5.4: Integer Types (http://www.ada-auth.org/standards/12rm/html/RM-3-5-4.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-3-5-4.html)]

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 actually a 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 enumeration 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';
```

```
type Day_Of_Week is (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday);

subtype Working_Day is Day_Of_Week range Monday .. Friday;
```

Using enumerations

Enumeration types being scalar subtypes, type attributes such as `First` and `Succ` will allow stepping through a subsequence of the values.

```
   case Day_Of_Week'First is
      when Sunday =>
         ISO (False);
      when Day_Of_Week'Succ (Sunday) =>
         ISO (True);
      when Tuesday .. Saturday =>
         raise Program_Error;
   end case;
```

A loop will automatically step through the values of the subtype's range. Filtering week days to include only working days with an even position number:

```
   for Day in Working_Day loop
      if Day_Of_Week'Pos (Day) mod 2 = 0 then
         Work_In_Backyard;
      end if;
   end loop;
```

Enumeration types can be used as array index subtypes, yielding a table feature:

```
type Officer_ID is range 0 .. 50;
type Schedule is array (Working_Day) of Officer_ID;
```

See also

Wikibook

- Ada Programming
- Ada Programming/Types
- Ada Programming/Libraries/Standard
- Ada_Programming/Attributes/'First
- Ada_Programming/Attributes/'Last
- Ada_Programming/Attributes/'Pred
- Ada_Programming/Attributes/'Succ
- Ada_Programming/Attributes/'Img
- Ada_Programming/Attributes/'Image
- Ada_Programming/Attributes/'Value
- Ada_Programming/Attributes/'Pos
- Ada_Programming/Attributes/'Val
- Ada_Programming/Attributes/'Enum_Rep

- Ada_Programming/Attributes/'Enum_Val

Ada Reference Manual

- 3.5.1 Enumeration Types (http://www.adaic.com/standards/05rm/html/RM-3-5-1.html) (Annotated (http://www.adaic.com/standards/05aarm/html/AA-3-5-1.html))

Floating point types

Description

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 (http://www.adaic.com/standards/05rm/html/RM-3-5-7.html) (Annotated (http://www.adaic.com/standards/05aarm/html/AA-3-5-7.html))

Fixed point types

Description

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 integer 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 an integer 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 1.0 / (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 — the 0° and 360° mark 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 an integer 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.

```
type Angle is delta Pi/2.0**31 range -Pi .. Pi;
for Angle'Small use Pi/2.0**31;
```

As internal representation, you will get a 32 bit signed integer type.

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 integer power of 10 — otherwise the declaration is illegal.

```
delta 10.0**(+2) digits 12
delta 10.0**(-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 an integer power of 10 as 'Small. The following two declarations are equivalent with respect to the internal representation:

```
-- decimal fixed point

type Duration is delta 10.0**(-9) digits 9;
```

```
-- ordinary fixed point

type Duration is delta 10.0**(-9) range -1.0 .. 1.0;
for Duration'Small use 10.0**(-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.
- Truncation is required for decimal, not for ordinary, fixed point in multiplication and division (RM 4.5.5: (21) (http://www.adaic.com/standards/95lrm/html/RM-4-5-5.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-4-5-5.html)]) and type conversions. Operations on decimal fixed point are fully specified, which is not true for ordinary fixed point.
- The following attributes are only defined for decimal fixed point: T'Digits (RM 3.5.10: (10) (http://www.adaic.com/standards/95lrm/html/RM-3-5-10.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-3-5-10.html)]) corresponds to the number of decimal digits that are representable; T'Scale (RM 3.5.10: (11) (http://www.adaic.com/standards/95lrm/html/RM-3-5-10.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-3-5-10.html)], taken from **COBOL**) indicates the position of the point relative to the rightmost significant digits; T'Round (RM 3.5.10: (12) (http://www.adaic.com/standards/95lrm/html/RM-3-5-10.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-3-5-10.html)]) can be used to specify rounding on conversion.
- Package Decimal (RM F.2 (http://www.adaic.com/standards/95lrm/html/RM-F-2.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-F-2.html)]), 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) (http://www.adaic.com/standards/95lrm/html/RM-A-10-1.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-A-10-1.html)]) has semantics different from Fixed_IO (RM A.10.1: (68) (http://www.adaic.com/standards/95lrm/html/RM-A-10-1.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-A-10-1.html)]).
- 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 (http://www.adaic.com/standards/95lrm/html/RM-3-5-9.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-3-5-9.html)]

Ada 2005 Reference Manual

- 3.5.9: Fixed Point Types (http://www.adaic.com/standards/05rm/html/RM-3-5-9.html) [Annotated (http://www.adaic.com/standards/05aarm/html/AA-3-5-9.html)]

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 a specific letter appears inside a text, you could use:

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

Very often, the index does not have semantic contents by itself, it is just used a means to identify elements for instance in a list. Thus, as a general advice, do not use negative indices in these cases. 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.

There are, however, cases, where negative indices make sense. So use indices adapted to the problem at hand. Imagine you are a chemist doing some experiments depending on the temperature:

```
type Temperature is range -10 .. +40;  -- Celsius

type Experiment is array (Temperature ) of Something;
```

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 some languages you would resort to pointers to element types. 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 demand 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 than 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 slice assignments are done in one go, not in a loop character by character, so that assignments with overlapping ranges work as expected:

```
Name: String (1 .. 13) := "Lady Ada     ";

Name (6 .. 13) := Name (1 .. 8);
```

The result is "Lady Lady Ada" (and not "Lady Lady Lad").

Slicing

As has been shown above, in slice assignments index ranges slide. Also subtype conversions make index ranges slide:

```
subtype Str_1_8 is String (1 .. 8);
```

The result of Str_1_8 (Name (6 .. 13)) has the new bounds 1 and 8 and contents "Lady Ada" and is not a copy. This is the best way to change the bounds of an array or of parts thereof.

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 chosen to show a few common beginner's mistakes:

1. The assumption that strings begin with the index value 1 is wrong (cf. World'First = 7 on the second line).
2. The assumption (which follows from the first one) that X'Length = X'Last is wrong.
3. The assumption that X'Last >= X'First; this is not true for empty strings.

The index subtype of predefined type String is Positive, therefore excluding 0 or -17 etc. from the set of possible index values, by subtype constraint (of Positive). Also, 'A' or 2.17e+4 are excluded, since they are not of type Positive.

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 for loop on arrays. When looping over all elements of an array, you need not know the actual index range; by using the attribute, one of the most frequent errors, accessing elements out of the index range, can never occur:

```
for I in World'Range loop
  ... World (I)...
end loop;
```

'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. Any array whose last index value is lower than the first index value is empty. 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);

Also_Empty: Some_Array (42 .. 10);
```

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 there are 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 (http://www.adaic.com/standards/95lrm/html/RM-3-6.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-3-6.html)]

Ada 2005 Reference Manual

- 3.6: Array Types (http://www.adaic.com/standards/05rm/html/RM-3-6.html) [Annotated (http://www.adaic.com/standards/05aarm/html/AA-3-6.html)]

Ada Quality and Style Guide

- 10.5.7 Packed Boolean Array Shifts (http://www.adaic.org/docs/95style/html/sec_10/10-5-7.html)

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 (http://www.adacore.com/2007/05/14/gem-1/), 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);
```

In order to access a component of a record instance, use the dot delimiter (.), as in `BMW.Number_Wheels`.

Discriminated record

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

...

Item : Discriminated_Record := (Size => Value'Length, A => Value);
```

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
            Timeout    : Duration := 0.0;
         when Green =>
            -- components for green
            Whatever : Positive := 1;
      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
                            Timeout => 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
            Timeout    : Duration;
         when Green =>
            -- components for green
            Whatever : Positive := 1;
      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 from Ada 2005 on.

```
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 one abstract primitive operation, i.e. one of its operations is not defined and implementation must be provided by derivatives of the abstract type.

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 (http://www.adaic.com/standards/95lrm/html/RM-3-8.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-3-8.html)]

Ada 2005

- 3.8: Record Types (http://www.adaic.com/standards/05rm/html/RM-3-8.html) [Annotated (http://www.adaic.com/standards/05aarm/html/AA-3-8.html)]
- B.3.3: Pragma Unchecked_Union (http://www.adaic.com/standards/05rm/html/RM-B-3-3.html) [Annotated (http://www.adaic.com/standards/05aarm/html/AA-B-3-3.html)]

Ada Issues

- AI95-00216-01 Unchecked unions — variant records with no run-time discriminant (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00216.TXT)

Access types

What's an Access Type?

Access types in Ada are what other languages call pointers. They point to objects located at certain addresses. So normally one can think of access types as simple addresses (there are exceptions from this simplified view). Ada instead of saying *points to* talks of *granting access to* or *designating* an object.

Objects of access types are implicitly initialized with `null`, i.e. they point to nothing when not explicitly initialized.

Access types should be used rarely in Ada. In a lot of circumstances where pointers are used in other languages, there are other ways without pointers. If you need dynamic data structures, first check whether you can use the Ada Container library. Especially for indefinite record or array components, the Ada 2012 package Ada.Containers.Indefinite_Holders (RM A.18.18 (http://www.ada-auth.org/standards/12rm/html/RM-A-18-18.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-A-18-18.html)]) can be used instead of pointers.

There are four kinds of access types in Ada: Pool access types - General access types - Anonymous access types - Access to subprogram 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). A pointer of these types cannot point to a stack or library level (static) object or an object in a different storage pool. Therefore, conversion between pool access types is illegal. (Unchecked_Conversion may be used, but note that deallocation via an access object with a storage pool different from the one it was allocated with is erroneous.)

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

type Person_Access is access Person;
```

A storage size clause may be used to limit the corresponding (implementation defined anonymous) storage pool. A storage size clause of 0 disables calls of an allocator.

```
for Person_Access'Storage_Size use 0;
```

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

```
for Person_Access'Storage_Pool use Pool_Name;
```

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

```
Father: Person_Access := new Person;                                      -- uninitialized
Mother: Person_Access := new Person'(Mothers_First_Name, Mothers_Last_Name);  -- initialized
```

You access the object in the storage pool by appending `.all`. `Mother.all` is the complete record; components are denoted as usual with the dot notation: `Mother.all.First_Name`. When accessing components, *implicit dereferencing* (i.e. omitting `all`) can serve as a convenient shorthand:

```
Mother.all := (Last_Name => Father.Last_Name, First_Name => Mother.First_Name);  -- marriage
```

Implicit dereferencing also applies to arrays:

```
   type Vector is array (1 .. 3) of Complex;
   type Vector_Access is access Vector;

   VA: Vector_Access := new Vector;
   VB: array (1 .. 3) of Vector_Access := (others => new Vector);

   C1: Complex := VA (3);    -- a shorter equivalent for VA   .all (3)
   C2: Complex := VB (3)(1); -- a shorter equivalent for VB(3).all (1)
```

Be careful to discriminate between deep and shallow copies when copying with access objects:

```
Obj1.all := Obj2.all;  -- Deep copy: Obj1 still refers to an object different from Obj2, but it has the same content
Obj1 := Obj2;          -- Shallow copy: Obj1 now refers to the same object as Obj2
```

Deleting objects from a storage pool

Although the Ada standard mentions a garbage collector, which would automatically remove all unneeded objects that have been created on the heap (when no storage pool has been defined), only Ada compilers targeting a virtual machine like Java or .NET actually have garbage collectors.

When an access type goes out of scope, the corresponding still allocated data items are finalized (i.e. they do no longer exist) in an arbitrary order; the allocated memory, however, is only freed when the attribute Storage_Size has been defined for the access type via an attribute_definition clause. (Note: *Finalization* and *deallocation* are different things!)

Proof

The following are excerpts from the Ada Reference Manual. The ellipses stand for parts not relevant for the case.

RM 3.10(7/1) There are ... *access-to-object* types, whose values designate objects... Associated with an access-to-object type is a *storage pool*; several access types may share the same storage pool. ... A storage pool is an area of storage used to hold dynamically allocated objects (called *pool elements*) created by allocators.

(8) Access-to-object types are further subdivided into *pool-specific* access types, whose values can designate only the elements of their associated storage pool...

RM 7.6(1) ... Every object is finalized before being destroyed (for example, by leaving a subprogram_body containing an object_declaration, or by a call to an instance of Unchecked_Deallocation)...

RM 7.6.1(5) For the *finalization* of an object:

(6/3) If the full type of the object is an elementary type, finalization has no effect;

(7/3) If the full type of the object is a tagged type, and the tag of the object identifies a controlled type, the Finalize procedure of that controlled type is called;

(10) Immediately before an instance of Unchecked_Deallocation reclaims the storage of an object, the object is finalized. If an instance of Unchecked_Deallocation is never applied to an object created by an allocator, the object will still exist when the corresponding master completes, and it will be finalized then.

(11.1/3) Each nonderived access type *T* has an associated *collection*, which is the set of objects created by allocators of *T*, or of types derived from *T*. Unchecked_Deallocation removes an object from its collection. Finalization of a collection consists of finalization of each object in the collection, in an arbitrary order...

RM 13.11(1) Each access-to-object type has an associated storage pool. The storage allocated by an allocator comes from the pool; instances of Unchecked_Deallocation return storage to the pool. Several access types can share the same pool.

(2/2) A storage pool is a variable of a type in the class rooted at Root_Storage_Pool, which is an abstract limited controlled type. By default, the implementation chooses a *standard storage pool* for each access-to-object type...

(11) A *storage pool type* (or *pool type*) is a descendant of Root_Storage_Pool. The *elements* of a storage pool are the objects allocated in the pool by allocators.

(15) Storage_Size or Storage_Pool may be specified for a nonderived access-to-object type via an attribute_definition_clause...

(17) If Storage_Pool is not specified for a type defined by an access_to_object_definition, then the implementation chooses a standard storage pool for it in an implementation-defined manner...

(18/4) If Storage_Size is specified for an access type *T*, an implementation-defined pool *P* is used for the type. The Storage_Size of *P* is at least that requested, and the storage for *P* is reclaimed when the master containing the declaration of the access type is left...

Example

The following program will compile but will fail the accessibility check on runtime with an exception.

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

procedure Main is
   function Accessibility_Check_Fail
     return access String
   is
      -- Declare a new access type locally.
      -- All memory with this type will be finalized but not freed
      -- when the this type goes out of scope.
      type A_Type is access String;  -- no Storage_Size defined

      X : A_Type := new String'("x");  -- storage will be lost
      Y : access String;  -- defined locally
   begin
      Y := X;  -- data defined in a local pool will be finalized when function returns
      return Y;  -- exception should be raised
   end Accessibility_Check_Fail;
begin
   -- Accessibility check will fail because the accessiblity level associated
   -- with Y is deeper than the accessibility level of this scope.
   Put_Line(Accessibility_Check_Fail.all);
end Main;
```

~~There is also a `pragma Controlled`, which, when applied to such an access type, prevents automatic garbage collection of objects created with it.~~ Note that `pragma Controlled` was dropped from Ada 2012, subpools for storage management replacing it. See RM 2012 13.11.3 (http://www.ada-auth.org/standards/12rm/html/RM-13-11-3.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-13-11-3.html)] and 13.11.4 (http://www.ada-auth.org/standards/12rm/html/RM-13-11-4.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-13-11-4.html)].

Therefore, in order to delete an object from the heap, you need the generic unit Ada.Unchecked_Deallocation. Apply utmost care to not create dangling pointers when deallocating objects as is shown in the example below. (And note that deallocating objects with a different access type than the one with which they were created is erroneous when the corresponding storage pools are different.)

```
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;
```

It is exactly because of this problem with dangling pointers that the deallocation operation is called **unchecked**. It is the chore of the programmer to take care that this does not happen.

Since Ada allows for user-defined storage pools, you could also try a garbage collector library.

Constructing Reference Counting Pointers

You can find some implementations of reference counting pointers, called *Safe* or *Smart Pointers*, on the net. Using such a type prevents caring about deallocation, since this will automatically be done when there are no more pointers to an object. But be careful - most of those implementations do not prevent deliberate deallocation, thus undermining the alleged safety attained with their use.

A nice tutorial how to construct such a type can be found in a series of Gems on the AdaCore web site.

Gem #97: Reference Counting in Ada – Part 1 (https://www.adacore.com/gems/gem-97-reference-counting-in-ada-part-1) This little gem constructs a simple reference counted pointer that does not prevent deallocation, i.e. is inherently unsafe.

Gem #107: Preventing Deallocation for Reference-counted Types (https://www.adacore.com/gems/gem-107-preventing-deallocation-for-reference-counted-types) This further gem describes how to arrive at a pointer type whose safety cannot be compromised (tasking issues aside). The cost of this improved safety is awkward syntax.

Gem #123: Implicit Dereferencing in Ada 2012 (https://www.adacore.com/gems/gem-123-implicit-dereferencing-in-ada-2012/) This gem shows how to simplify the syntax with the new Ada 2012 generation. (Admittedly, this gem is a bit unrelated to reference counting since the new language feature can be applied to any kind of container.)

General access

General access types grant access to objects created on any storage pool, on the stack or at library level (static). They come in two versions, granting either read-write access or read-only access. Conversions between general access types are allowed, but subject to certain access level checks.

Dereferencing is like for pool access types. Objects (other than pool objects) to be referenced have to be declared `aliased`, and references to them are created with the attribute `'Access`. Access level restrictions prevent accesses to objects from outliving the accessed object, which would make the program erroneous. The attribute `'Unchecked_Access` omits the corresponding checks.

Access to Variable

When the keyword `all` is used in their definition, they grant read-write access.

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

Access to Constant

General access types granting read-only access to the referenced object use the keyword `constant` in their definition. The referenced object may be a constant or a variable.

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

Some examples

```
 type General_Pointer  is access all      Integer;
 type Constant_Pointer is access constant Integer;

 I1: aliased constant Integer := 10;
 I2: aliased Integer;

 P1: General_Pointer  := I1'Access;  -- illegal
 P2: Constant_Pointer := I1'Access;  -- OK, read only
 P3: General_Pointer  := I2'Access;  -- OK, read and write
 P4: Constant_Pointer := I2'Access;  -- OK, read only

 P5: constant General_Pointer := I2'Access;  -- read and write only to I2
```

Anonymous access

Also *Anonymous access types* come in two versions like general access types, granting either read-write access or read-only access depending on whether the keyword `constant` appears.

An anonymous access can be used as a parameter to a subprogram or as a discriminant. Here are some examples:

```
procedure Modify (Some_Day: access          Day_Of_Month);
procedure Test   (Some_Day: access constant Day_Of_Month);  -- Ada 2005 only
```

```
task type Thread (Execute_For_Day: access Day_Of_Month) is
   ...
end Thread;
```

```
type Day_Data (Store_For_Day: access Day_Of_Month) is record
  -- components
end record;
```

Before using an anonymous access, you should consider a named access type or, even better, consider if the "**out**" or "**in out**" modifier is not more appropriate.

This language feature is only available from Ada 2005 on.

In Ada 2005, anonymous accesses are allowed in more circumstances:

```
type Object is record
  M   : Integer;
  Next: access Object;
end record;

X: access Integer;

function F return access constant Float;
```

Implicit Dereference

This language feature has been introduced in Ada 2012.

Ada 2012 simplifies accesses to objects via pointers with new syntax.

Imagine you have a container holding some kind of elements.

```
type Container   is private;
type Element_Ptr is access Element;

procedure Put (X: Element; Into: in out Container);
```

Now, how do you access elements stored in the container. Of course, you can retrieve them by

```
function Get (From: Container) return Element;
```

This will however copy the element, which is unfortunate if the element is big. You get direct access with

```
function Get (From: Container) return Element_Ptr;
```

Now, pointers are dangerous since you might easily create dangling pointers like so:

```
P: Element_Ptr := Get (Cont);
P.all := E;
Free (P);
... Get (Cont) -- this is now a dangling pointer
```

Use of an accessor object instead of an access type can prevent inadvertent deallocation (this is still Ada 2005):

```
type Accessor (Data: not null access Element) is limited private;  -- read/write access
function Get (From: Container) return Accessor;
```

(For the null exclusion **not null** in the declaration of the discriminant, see below). Access via such an accessor is safe: The discriminant can only be used for dereferencing, it cannot be copied to an object of type Element_Ptr because its accessibility level is deeper. In the form above, the accessor provides read and write access. If the keyword **constant** is added, only read access is possible.

```
type Accessor (Data: not null access constant Element) is limited private;  -- only read access
```

Access to the container object now looks like so:

```
Get (Cont).all      := E;  -- via access type: dangerous
Get (Cont).Data.all := E;  -- via accessor: safe, but ugly
```

Here the new Ada 2012 feature of *aspects* comes along handy; for the case at hand, the aspect *Implicit_Dereference* is the one we need:

```
type Accessor (Data: not null access Element) is limited private
   with Implicit_Dereference => Data;
```

Now rather than writing the long and ugly function call of above, we can just omit the discriminant and its dereference like so:

```
Get (Cont).Data.all := E;  -- Ada 2005 via accessor: safe, but ugly
Get (Cont)          := E;  -- Ada 2012 implicit dereference
```

Note that the call `Get (Cont)` is overloaded — it can denote the accessor object or the element, the compiler will select the correct interpretation depending on context.

Null exclusions

This language feature is only available from Ada 2005 on.

All access subtypes can be modified with **`not null`**, objects of such a subtype can then never have the value null, so initializations are compulsory.

```
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 language also allows to declare *the first subtype* directly with a null exclusion:

```
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 renders objects of this type nearly unusable (for example, you are unable to free the allocated memory). Not null accesses are intended for access *subtypes*, object *declarations*, and subprogram *parameters*.[7] (https://groups.google.com/group/comp.lang.ada/msg/13a41ced7af75192)

Access to Subprogram

An access to subprogram allows the caller 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 from Ada 2005 on.

```
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.

Access FAQ

A few "Frequently Asked Question" and "Frequently Encountered Problems" (mostly from 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 simple `access` at all?" - And indeed some programmers never use simple `access`.

Unchecked_Deallocation is always dangerous if misused. It is just as easy to deallocate a pool-specific object twice, and just as dangerous as deallocating a stack object. The advantage of "access all" is that you may not need to use Unchecked_Deallocation at all.

Moral: if you have (or may have) a valid reason to store an 'Access or 'Unchecked_Access into an access object, then use "access all" and don't worry about it. If not, the mantra of "least privilege" suggests that the "all" should be left out (don't enable capabilities that you are not going to use).

The following (perhaps disastrous) example 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. The reason is that `access` does not allow pointers to be created from stack objects.

Access vs. System.Address

An access can be something different from a mere memory address, it may be 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, use 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 will 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 C would have used them as well. Of course the RM contains precise rules on when to use a pointer for "in", "out", and "in out" - see "B.3: Interfacing with C (http://www.ada-auth.org/standards/12rm/html/RM-B-3.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-B-3.html)]".

Where is void*?

While actually a problem for "interfacing with C", here are 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);
```

Thin and Fat Access Types

The difference between an access type and an address will be detailed in the following. The term *pointer* is used because this is usual terminology.

There is a predefined unit `System.Address_to_Access_Conversion` converting back and forth between access values and addresses. Use these conversions with care, as is explained below.

Thin Pointers

Thin pointers grant access to constrained subtypes.

```
type Int     is range -100 .. +500;
type Acc_Int is access Int;

type Arr     is array (1 .. 80) of Character;
type Acc_Arr is access Arr;
```

Objects of subtypes like these have a static size, so a simple address suffices to access them. In the case of arrays, this is generally the address of the first element.

For pointers of this kind, use of `System.Address_to_Access_Conversion` is safe.

Fat Pointers

```
type Unc     is array (Integer range <>) of Character;
type Acc_Unc is access Unc;
```

Objects of subtype `Unc` need a constraint, i.e. a start and a stop index, thus pointers to them need also to include those. So a simple address like the one of the first component is not sufficient. Note that A'Address is the same as A(A'First)'Address for any array object.

For pointers of this kind, `System.Address_to_Access_Conversion` will probably not work satisfactorily.

Example

```
CO: aliased Unc (-1 .. +1) := (-1 .. +1 => ' ');
UO: aliased Unc            := (-1 .. +1 => ' ');
```

Here, CO is a *nominally constrained* object, a pointer to it need not store the constraint, i.e. a thin pointer suffices. In contrast, UO is an object of a *nominally unconstrained* subtype, its *actual subtype* is constrained by the initial value.

```
A: Acc_Unc            := CO'Access;  -- illegal
B: Acc_Unc            := UO'Access;  -- OK
C: Acc_Unc (CO'Range) := CO'Access;  -- also illegal
```

The relevant paragraphs in the RM are difficult to understand. In short words:

An access type's target type is called the *designated subtype*, in our example `Unc`. RM 3.10.2 (http://www.ada-auth.org/standards/12rm/html/RM-3-10-2.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-3-10-2.html)](27.1/2) requires that `Acc_Unc`'s designated subtype statically match the *nominal subtype* of the object.

Now the nominal subtype of `C0` is the constrained anonymous subtype `Unc (-1 .. +1)`, the nominal subtype of `U0` is the unconstrained subtype `Unc`. In the illegal cases, the designated and nominal subtypes do not statically match.

See also

Wikibook

- Ada Programming
- Ada Programming/Types

Ada Reference Manual

Ada 95

- 4.8: Allocators (http://www.adaic.com/standards/95lrm/html/RM-4-8.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-4-8.html)]
- 13.11: Storage Management (http://www.adaic.com/standards/95lrm/html/RM-13-11.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-13-11.html)]
- 13.11.2: Unchecked Storage Deallocation (http://www.adaic.com/standards/95lrm/html/RM-13-11-2.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-13-11-2.html)]
- 3.7: Discriminants (http://www.adaic.com/standards/95lrm/html/RM-3-7.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-3-7.html)]
- 3.10: Access Types (http://www.adaic.com/standards/95lrm/html/RM-3-10.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-3-10.html)]
- 6.1: Subprogram Declarations (http://www.adaic.com/standards/95lrm/html/RM-6-1.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-6-1.html)]
- B.3: Interfacing with C (http://www.adaic.com/standards/95lrm/html/RM-B-3.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-B-3.html)]

Ada 2005

- 4.8: Allocators (http://www.adaic.com/standards/05rm/html/RM-4-8.html) [Annotated (http://www.adaic.com/standards/05aarm/html/AA-4-8.html)]
- 13.11: Storage Management (http://www.adaic.com/standards/05rm/html/RM-13-11.html) [Annotated (http://www.adaic.com/standards/05aarm/html/AA-13-11.html)]

- 13.11.2: Unchecked Storage Deallocation (http://www.adaic.com/standards/05rm/html/RM-13-11-2.html) [Annotated (http://www.adaic.com/standards/05aarm/html/AA-13-11-2.html)]
- 3.7: Discriminants (http://www.adaic.com/standards/05rm/html/RM-3-7.html) [Annotated (http://www.adaic.com/standards/05aarm/html/AA-3-7.html)]
- 3.10: Access Types (http://www.adaic.com/standards/05rm/html/RM-3-10.html) [Annotated (http://www.adaic.com/standards/05aarm/html/AA-3-10.html)]
- 6.1: Subprogram Declarations (http://www.adaic.com/standards/05rm/html/RM-6-1.html) [Annotated (http://www.adaic.com/standards/05aarm/html/AA-6-1.html)]
- B.3: Interfacing with C (http://www.adaic.com/standards/05rm/html/RM-B-3.html) [Annotated (http://www.adaic.com/standards/05aarm/html/AA-B-3.html)]

Newest RM

- 3.10: Access Types (http://www.ada-auth.org/standards/12rm/html/RM-3-10.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-3-10.html)]
- 7.6: Assignment and Finalization (http://www.ada-auth.org/standards/12rm/html/RM-7-6.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-7-6.html)]
- 7.6.1: Completion and Finalization (http://www.ada-auth.org/standards/12rm/html/RM-7-6-1.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-7-6-1.html)]
- 13.11: Storage Management (http://www.ada-auth.org/standards/12rm/html/RM-13-11.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-13-11.html)]

Ada Quality and Style Guide

- 5.4.5 Dynamic Data (http://www.adaic.org/docs/95style/html/sec_5/5-4-5.html)
- 5.9.2 Unchecked Deallocation (http://www.adaic.org/docs/95style/html/sec_5/5-9-2.html)

References

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 (http://www.adacore.com/2007/05/14/gem-1/) 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 (http://www.adaic.com/standards/05rm/html/RM-6-5.html) [Annotated (http://www.adaic.com/standards/05aarm/html/AA-6-5.html)]) 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 this declaration of two objects, the initial expression will be evaluated twice and can deliver different values, because it is equivalent to the sequence:[1]

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

So X is initialised before Y.

See also

Ada 95 Reference Manual

- 7.5: Limited Types (http://www.adaic.com/standards/95lrm/html/RM-7-5.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-7-5.html)]

Ada 2005 Reference Manual

- 7.5: Limited Types (http://www.adaic.com/standards/05rm/html/RM-7-5.html) [Annotated (http://www.adaic.com/standards/05aarm/html/AA-7-5.html)]

Ada Quality and Style Guide

- 5.3.3 Private Types (http://www.adaic.org/docs/95style/html/sec_5/5-3-3.html)
- 8.3.3 Formal Private and Limited Private Types (http://www.adaic.org/docs/95style/html/sec_8/8-3-3.html)

References

1. ISO/IEC 8652:2007. "3.3.1 Object Declarations (7)". *Ada 2005 Reference Manual*. "Any declaration [...] with more than one `defining_identifier` is equivalent to a series of declarations each containing one `defining_identifier` from the list, [...] in the same order as the list." {{cite book}}: Unknown parameter |chapterurl= ignored (|chapter-url= suggested) (help)

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 (the predefined type String) are arrays of Character, and consequently of a fixed length. Since 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 string has been created the length stays constant. Try the following program which shows a typical mistake:

File: show_commandline_1.adb (view (https://sourceforge.net/p/wikibook-ada/code/HEAD/tree/trunk/demos/Source/show_commandline_1.adb), plain text (https://sourceforge.net/p/wikibook-ada/code/HEAD/tree/trunk/demos/Source/show_commandline_1.adb?format=raw), download page (https://sourceforge.net/project/showfiles.php?group_id=124904), browse all (https://wikibook-ada.sourceforge.net/samples))

```
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 Handling 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 (https://sourceforge.net/p/wikibook-ada/code/HEAD/tree/trunk/demos/Source/show_commandline_2.adb), plain text (https://sourceforge.net/p/wikibook-ada/code/HEAD/tree/trunk/demos/Source/show_commandline_2.adb?format=raw), download page (https://sourceforge.net/project/showfiles.php?group_id=124904), browse all (https://wikibook-ada.sourceforge.net/samples))

```
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 number 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 (https://sourceforge.net/p/wikibook-ada/code/HEAD/tree/trunk/demos/Source/show_commandline_3.adb), plain text (https://sourceforge.net/p/wikibook-ada/code/HEAD/tree/trunk/demos/Source/show_commandline_3.adb?format=raw), download page (https://sourceforge.net/project/showfiles.php?group_id=124904), browse all (https://wikibook-ada.sourceforge.net/samples))

```
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. This is because Unbounded_String type is implemented using dynamic memory allocation behind the scenes, providing lower efficiency but maximum flexibility.

File: show_commandline_4.adb (view (https://sourceforge.net/p/wikibook-ada/code/HEAD/tree/trunk/demos/Source/show_commandline_4.adb), plain text (https://sourceforge.net/p/wikibook-ada/code/HEAD/tree/trunk/demos/Source/show_commandline_4.adb?format=raw), download page (https://sourceforge.net/project/showfiles.php?group_id=124904), browse all (https://wikibook-ada.sourceforge.net/samples))

```
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 (disregarding the buggy first example) — this makes using Unbounded-Length Strings very appealing.

See also

Wikibook

- Ada Programming

Ada 95 Reference Manual

- 2.6: String Literals (http://www.adaic.com/standards/95lrm/html/RM-2-6.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-2-6.html)]
- 3.6.3: String Types (http://www.adaic.com/standards/95lrm/html/RM-3-6-3.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-3-6-3.html)]
- A.4.3: Fixed-Length String Handling (http://www.adaic.com/standards/95lrm/html/RM-A-4-3.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-A-4-3.html)]

- A.4.4: Bounded-Length String Handling (http://www.adaic.com/standards/95lrm/html/RM-A-4-4.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-A-4-4.html)]
- A.4.5: Unbounded-Length String Handling (http://www.adaic.com/standards/95lrm/html/RM-A-4-5.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-A-4-5.html)]

Ada 2005 Reference Manual

- 2.6: String Literals (http://www.adaic.com/standards/05rm/html/RM-2-6.html) [Annotated (http://www.adaic.com/standards/05aarm/html/AA-2-6.html)]
- 3.6.3: String Types (http://www.adaic.com/standards/05rm/html/RM-3-6-3.html) [Annotated (http://www.adaic.com/standards/05aarm/html/AA-3-6-3.html)]
- A.4.3: Fixed-Length String Handling (http://www.adaic.com/standards/05rm/html/RM-A-4-3.html) [Annotated (http://www.adaic.com/standards/05aarm/html/AA-A-4-3.html)]
- A.4.4: Bounded-Length String Handling (http://www.adaic.com/standards/05rm/html/RM-A-4-4.html) [Annotated (http://www.adaic.com/standards/05aarm/html/AA-A-4-4.html)]
- A.4.5: Unbounded-Length String Handling (http://www.adaic.com/standards/05rm/html/RM-A-4-5.html) [Annotated (http://www.adaic.com/standards/05aarm/html/AA-A-4-5.html)]

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 three 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 – with a caveat, see Ada Programming/Constants. This is the default when no mode is given. The actual parameter can be 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.)

A parameter of any mode may also be explicitly `aliased`.

`access`
: The formal parameter is an access (a pointer) to some variable. (This is not a parameter mode from the reference manual point of view.)

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 (which you can assume to do what is most sensible). Tagged types are always passed by reference.

Explicitly `aliased` parameters and `access` parameters specify pass by reference.

Unlike in the C class of programming languages, Ada subprogram calls cannot have empty parameters parentheses `( )` when there are 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 uninitialized 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 should 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. Until Ada 2005, 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. Ada 2012 has removed this restriction.

Here is an example of a function body:

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

(There is, by the way, also the attribute `Integer'Min`, see RM 3.5.) Or in Ada2012:

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

or even shorter as an *expression function*

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

The formal parameters with mode **in** 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 especially 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 parameter; 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.

Renaming

Subprograms may be renamed. The parameter and result profile for a renaming-as-declaration must be mode conformant.

```
procedure Solve
  (A, B, C: in  Float;
   R1, R2 : out Float;
   Valid  : out Boolean) renames Quadratic_Equation;
```

This may be especially comfortable for tagged types.

```
package Some_Package is
  type Message_Type is tagged null record;
  procedure Print (Message: in Message_Type);
end Some_Package;
```

```
with Some_Package;
procedure Main is
  Message: Some_Package.Message_Type;
  procedure Print renames Message.Print;  -- this has convention intrinsic, see RM 6.3.1(10.1/2)
  Method_Ref: access procedure := Print'Access;  -- thus taking 'Access should be illegal; GNAT GPL 2012 allows this
begin  -- All these calls are equivalent:
```

```
   Some_Package.Print (Message);  -- traditional call without use clause
   Message.Print;                 -- Ada 2005 method.object call - note: no use clause necessary
   Print;                         -- Message.Print is a parameterless procedure and can be renamed as such
   Method_Ref.all;                -- GNAT GPL 2012 allows illegal call via an access to the renamed procedure Print
                                  -- This has been corrected in the current version (as of Nov 22, 2012)
end Main;
```

But note that `Message.Print'Access;` is illegal, you have to use a renaming declaration as above.

Since only mode conformance is required (and not full conformance as between specification and body), parameter names and default values may be changed with renamings:

```
procedure P (X: in Integer :=  0);
procedure R (A: in Integer := -1) renames P;
```

See also

Wikibook

- Ada Programming
- Ada Programming/Operators

Ada 95 Reference Manual

- Section 6: Subprograms (http://www.adaic.com/standards/95lrm/html/RM-6.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-6.html)]
- 4.4: Expressions (http://www.adaic.com/standards/95lrm/html/RM-4-4.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-4-4.html)]

Ada 2005 Reference Manual

- Section 6: Subprograms (http://www.adaic.com/standards/05rm/html/RM-6.html) [Annotated (http://www.adaic.com/standards/05aarm/html/AA-6.html)]

- 4.4: Expressions (http://www.adaic.com/standards/05rm/html/RM-4-4.html) [Annotated (http://www.adaic.com/standards/05aarm/html/AA-4-4.html)]

Ada Quality and Style Guide

- 4.1.3 Subprograms (http://www.adaic.org/docs/95style/html/sec_4/4-1-3.html)

Packages

Ada encourages the division of code into separate modules called *packages*. Each package can contain any combination of items.

Some of the benefits of using packages are:

- package contents are placed in a separate namespace, preventing naming collisions,
- implementation details of the package can be hidden from the programmer (information hiding),
- object orientation requires defining a type and its primitive subprograms within a package, and
- packages that are library units can be separately compiled.

Some of the more common package usages are:

- a group of related subprograms along with their shared data, with the data not visible outside the package,
- one or more data types along with subprograms for manipulating those data types, and
- a generic package that can be instantiated under varying conditions.

The following is a quote from the current Ada Reference Manual Section 7: Packages. RM 7(1) (http://www.ada-auth.org/standards/12rm/html/RM-7.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-7.html)]

> 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.

Separate compilation

Note: The following chapters deal with packages on *library level*. This is the most common use, since packages are the basic code structuring means in Ada. However, Ada being a block oriented language, packages can be declared at any level in any declarative region. In this case, the normal visibility rules apply, also for the package body.

Package specifications and bodies on library level are compilation units, so can be separately compiled. Ada has nothing to say about how and where compilation units are stored. This is implementation dependent. (Most implementations indeed store compilation units in files of their own; name suffixes vary, GNAT uses `.ads` and `.adb`, APEX `.1.ada` and `.2.ada`.) The package body can itself be divided into multiple parts by specifying that subprogram implementations or bodies of nested packages are **separate**. These are then further compilation units.

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 compilers determine the compilation sequence; no make file is needed. 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.

Parts of a package

A package generally consists of two parts, the specification and the body. A package specification can be further divided in two logical parts, the visible part and the private part. Only the visible part of the specification is mandatory. The private part of the specification is optional, and a package specification might not have a package body—the package body only exists to complete any *incomplete* items in the specification. Subprogram declarations are the most common *incomplete* items. There must not be a package body if there is no incomplete declaration, and there has to be a package body if there is some incomplete declaration in the specification.

To understand the value of the three-way division, consider the case of a package that has already been released and is in use. A change to the visible part of the specification will require that the programmers of all using software verify that the change does not affect the using code. A change to the private part of the declaration will require that all using code be recompiled but no review is normally needed. Some changes to the private part can change the meaning of the client code however. An example is changing a private record type into a private access type. This

change can be done with changes in the private part, but change the semantic meaning of assignment in the clients code. A change to the package body will only require that the file containing the package body be recompiled, because *nothing* outside of the package body can ever access anything within the package body (beyond the declarations in the specification part).

A common usage of the three parts is to declare the existence of a type and some subprograms that operate on that type in the visible part, define the actual structure of the type (e.g. as a record) in the private part, and provide the code to implement the subprograms in the package body.

The package specification — the visible part

The visible part of a package specification 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;
```

Since Range_10 is an integer type, there are a lot of operations declared implicitly in this package.

The private part

The private part of a package serves two purposes:

- To complete the deferred 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;
```

Since the type is private, clients cannot make any use of it as long as there are no operations defined in the visible part.

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);  -- not a standard Ada pragma
   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 or ASM. A package will be modeling 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);—not a standard Ada pragma

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

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 from Ada 2005 on.

```
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 from Ada 2005 on.

The limited with can be used to represent two mutually dependent type (or more) located in separate packages.

```
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 extremely 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 nameable, 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

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.[1] 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 packages.

```
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 (http://www.adaic.com/standards/05rm/html/RM-10-1-2.html) (Annotated (http://www.adaic.com/standards/05aarm/html/AA-10-1-2.html))). Child packages thus give better control over package dependences. 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. For example, `E: Integer := D + N.X;`

See also

Wikibook

- Ada Programming

Wikipedia

- Module

Ada 95 Reference Manual

- Section 7: Packages (http://www.adaic.com/standards/95lrm/html/RM-7.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-7.html)]

Ada 2005 Reference Manual

- Section 7: Packages (http://www.adaic.com/standards/05rm/html/RM-7.html) [Annotated (http://www.adaic.com/standards/05aarm/html/AA-7.html)]

Input Output

Overview

The standard Ada libraries provide several Input/Output facilities, each one adapted to specific needs. Namely, the language defines the following dedicated packages:

- Text_IO
- Sequential_IO
- Direct_IO
- Stream_IO

The programmer must choose the adequate package depending on the application needs. For example, the following properties of the data handled by the application should be considered:

- **Data contents**: plain text, or binary data?
- **Accessing the data**: random access, or sequential access?
- **Medium**: data file, console, network/data-bus?
- **Data structure**: homogeneous file (sequence of the same data field), heterogeneous file (different data fields)?
- **Data format**: adherence to an existing data format, or the application can freely choose a new one?

For example, Stream_IO is very powerful and can handle complex data structures but can be heavier than other packages; Sequential_IO is lean and easy to use but cannot be used by applications requiring random data access; Text_IO can handle just textual data, but it is enough for handling the command-line console.

The following table gives some advices for choosing the more adequate one:

Simple heuristics for choosing an I/O package

Data access	Plain text	Binary data	
		Homogeneous	Heterogeneous
Sequential	Text_IO	Sequential_IO	Stream_IO
Random	Stream_IO	Direct_IO	Stream_IO

So the most important lesson to learn is choosing the right one. This chapter will describe more in detail these standard packages, explaining how to use them effectively. Besides these Ada-defined packages for general I/O operations each Ada compiler usually has other implementation-defined Input-Output facilities, and there are also other external libraries for specialized I/O needs like XML processing or interfacing with databases.

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.Text_IO;

procedure Main is
  Str  : String (1 .. 80);
  Last : Natural;
begin
  Ada.Text_IO.Get_Line (Str, Last);
  Ada.Text_IO.Put_Line (Str (1 .. Last));
end;
```

This example copies text from standard input to standard output when all lines are shorter than 80 characters, the string length. See package Text I/O how to deal with longer lines.

The package also contains several generic packages for converting numeric and enumeration types to character strings; there are child packages for handling Bounded and Unbounded strings, allowing the programmer to read and write different data types in the same file easily (there are ready-to-use instantiations of these generic packages for the Integer, Float, and Complex types). Finally, the same family of Ada.Text_IO packages (including the several children and instantiation packages) for the type Wide_Character and Wide_Wide_Character.

It is worth noting that the family of Text_IO packages provide some automatic text processing. For example, the Get procedures for numeric and enumeration types ignore white space at the beginning of a line (Get_Line for String does not present this behavior), or adding line and page terminators when closing the file. This is thus adequate for applications handling simple textual data, but users requiring direct management of text (e.g. raw access to the character encoding) must consider other packages like Sequential_IO.

See this tutorial for a longer example of using text I/O in Ada.

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.

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 attributes add additional control information to the file, like for example the 'First and 'Last attributes for 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.Streams
  - Ada Programming/Libraries/Ada.Streams.Stream_IO
  - Ada Programming/Libraries/Ada.Text_IO.Text_Streams
- Ada Programming/Libraries/Ada.Text_IO
  - Ada Programming/Libraries/Ada.Text_IO.Enumeration_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.Float_IO (nested package)
  - Ada Programming/Libraries/Ada.Text_IO.Fixed_IO (nested package)
  - Ada Programming/Libraries/Ada.Text_IO.Decimal_IO (nested package)
  - Ada Programming/Libraries/Ada.Text_IO.Bounded_IO
  - Ada Programming/Libraries/Ada.Text_IO.Unbounded_IO
  - Ada Programming/Libraries/Ada.Text_IO.Complex_IO (specialized needs annex)
  - Ada Programming/Libraries/Ada.Text_IO.Editing (specialized needs annex)
- Ada Programming/Libraries/Ada.Integer_Text_IO
- Ada Programming/Libraries/Ada.Float_Text_IO
- Ada Programming/Libraries/Ada.Complex_Text_IO (specialized needs annex)
- Ada Programming/Libraries/Ada.Storage_IO (not a general-purpose I/O package)
- Ada Programming/Libraries/Ada.IO_Exceptions
- Ada Programming/Libraries/Ada.Command_Line

- Ada Programming/Libraries/Ada.Directories
- Ada Programming/Libraries/Ada.Environment_Variables
- Ada Programming/Libraries/GNAT.IO (implementation defined)
- Ada Programming/Libraries/GNAT.IO_Aux (implementation defined)
- Ada Programming/Libraries/GNAT.Calendar.Time_IO (implementation defined)
- Ada Programming/Libraries/System.IO (implementation defined)
- Ada Programming/Libraries
  - Ada Programming/Libraries/GUI
  - Ada Programming/Libraries/Web
  - Ada Programming/Libraries/Database
- Ada Programming/Platform
- Ada Programming/Platform/Linux
- Ada Programming/Platform/Windows

Ada Reference Manual

- A.6 Input-Output (http://www.adaic.com/standards/05rm/html/RM-A-6.html) (Annotated (http://www.adaic.com/standards/05aarm/html/AA-A-6.html))
- A.7 External Files and File Objects (http://www.adaic.com/standards/05rm/html/RM-A-7.html) (Annotated (http://www.adaic.com/standards/05aarm/html/AA-A-7.html))
- A.8 Sequential and Direct Files (http://www.adaic.com/standards/05rm/html/RM-A-8.html) (Annotated (http://www.adaic.com/standards/05aarm/html/AA-A-8.html))
- A.10 Text Input-Output (http://www.adaic.com/standards/05rm/html/RM-A-10.html) (Annotated (http://www.adaic.com/standards/05aarm/html/AA-A-10.html))
- A.11 Wide Text Input-Output and Wide Wide Text Input-Output (http://www.adaic.com/standards/05rm/html/RM-A-11.html) (Annotated (http://www.adaic.com/standards/05aarm/html/AA-A-11.html))
- A.12 Stream Input-Output (http://www.adaic.com/standards/05rm/html/RM-A-12.html) (Annotated (http://www.adaic.com/standards/05aarm/html/AA-A-12.html))
- A.13 Exceptions in Input-Output (http://www.adaic.com/standards/05rm/html/RM-A-13.html) (Annotated (http://www.adaic.com/standards/05aarm/html/AA-A-13.html))

- A.14 File Sharing (http://www.adaic.com/standards/05rm/html/RM-A-14.html) (Annotated (http://www.adaic.com/standards/05aarm/html/AA-A-14.html))

Ada 95 Quality and Style Guide

- 7.7 Input/Output (http://www.adaic.org/docs/95style/html/sec_7/7-7.html)
  - 7.7.1 Name and Form Parameters (http://www.adaic.org/docs/95style/html/sec_7/7-7-1.html)
  - 7.7.2 File Closing (http://www.adaic.org/docs/95style/html/sec_7/7-7-2.html)
  - 7.7.3 Input/Output on Access Types (http://www.adaic.org/docs/95style/html/sec_7/7-7-3.html)
  - 7.7.4 Package Ada.Streams.Stream_IO (http://www.adaic.org/docs/95style/html/sec_7/7-7-4.html)
  - 7.7.5 Current Error Files (http://www.adaic.org/docs/95style/html/sec_7/7-7-5.html)

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  : constant 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: constant Float := 0.5 * (A + B + C);
   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 similarly to objects.

Ex.1 declares two exceptions:

```
Line_Failed, Line_Closed: exception;
```

However, exceptions 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.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;
```

Exception handlers

When an exception occurs, the normal flow of execution is abandoned and the exception is handed up the call sequence until a matching handler is found. Any declarative region (except a package specification) can have a handler. The handler names the exceptions it will handle. By moving up the call sequence, exceptions can become anonymous; in this case, they can only be handled with the `others` handler.

```
function F return Some_Type is
  ... -- declarations (1)
begin
  ... -- statements (2)
exception -- handlers start here (3)
  when Name_1 | Name_2 => ... -- The named exceptions are handled with these statements
  when others => ...  -- any other exceptions (also anonymous ones) are handled here
end F;
```

Exceptions raised in the declarative region itself (1) cannot be handled by handlers of this region (3); they can only be handled in outer scopes. Exceptions raised in the sequence of statements (2) can of course be handled at (3).

The reason for this rule is so that the handler can assume that any items declared in the declarative region (1) are well defined and may be referenced. If the handler at (3) could handle exceptions raised at (1), it would be unknown which items existed and which ones didn't.

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 (Exception_Message (Fail));
end;
```

Starting with Ada 2005, a simpler syntax can be used to associate a string message with exception occurrence.

```
-- This language feature is only available from Ada 2005 on.
declare
   Valve_Failure : exception;
begin
  ...
  raise Valve_Failure with "Failure while opening";
  ...
  raise Valve_Failure with "Failure while closing";
  ...
exception
  when Fail: Valve_Failure =>
    Put (Exception_Message (Fail));
end;
```

The Ada.Exceptions package also provides subprograms for saving exception occurrences and re-raising them.

See also

Wikibook

- Ada Programming

Ada 95 Reference Manual

- Section 11: Exceptions (http://www.adaic.com/standards/95lrm/html/RM-11.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-11.html)]
- 11.4.1: The Package Exceptions (http://www.adaic.com/standards/95lrm/html/RM-11-4-1.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-11-4-1.html)]

Ada 2005 Reference Manual

- Section 11: Exceptions (http://www.adaic.com/standards/05rm/html/RM-11.html) [Annotated (http://www.adaic.com/standards/05aarm/html/AA-11.html)]
- 11.4.1: The Package Exceptions (http://www.adaic.com/standards/05rm/html/RM-11-4-1.html) [Annotated (http://www.adaic.com/standards/05aarm/html/AA-11-4-1.html)]

Ada Quality and Style Guide

- **Chapter 4: Program Structure**
  - 4.3 Exceptions (http://www.adaic.org/docs/95style/html/sec_4/4-3.html)
    - 4.3.1 Using Exceptions to Help Define an Abstraction (http://www.adaic.org/docs/95style/html/sec_4/4-3-1.html)
- **Chapter 5: Programming Practices**
  - 5.8 Using Exceptions (http://www.adaic.org/docs/95style/html/sec_5/5-8.html)
    - 5.8.1 Handling Versus Avoiding Exceptions (http://www.adaic.org/docs/95style/html/sec_5/5-8-1.html)

Generics

Parametric polymorphism (generic units)

The idea of code reuse arises from the necessity for constructing large software systems combining well-established building blocks. The reusability of code improves the productivity and the quality of software. The generic units are one of the ways in which the Ada language supports this characteristic. A generic unit is a subprogram or package that defines algorithms in terms of types and operations that are not defined until the user instantiates them.

Note to C++ programmers: generic units are similar to C++ templates.

For example, to define a procedure for swapping 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 categories are acceptable as actuals. As a rule of thumb: the syntax expresses how the generic sees the type, i.e. it assumes the worst, not how the creator of the instance sees the type.

This is the syntax of RM 12.5 (http://www.ada-auth.org/standards/12rm/html/RM-12-5.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-12-5.html)]:

```
formal_type_declaration ::=
  type defining_identifier[discriminant_part] is formal_type_definition;

formal_type_definition ::= formal_private_type_definition
                         | formal_derived_type_definition
                         | formal_discrete_type_definition
                         | formal_signed_integer_type_definition
                         | formal_modular_type_definition
                         | formal_floating_point_definition
                         | formal_ordinary_fixed_point_definition
                         | formal_decimal_fixed_point_definition
                         | formal_array_type_definition
                         | formal_access_type_definition
                         | formal_interface_type_definition
```

This is quite complex, so some examples are given below. A type declared with the syntax `type T (<>)` denotes a type with *unknown discriminants*. This is the Ada vernacular for indefinite types, i.e. types for which objects cannot be declared without giving an initial expression. An example of such a type is one with a discriminant without default, another example is an unconstrained array type.

Generic formal type	Acceptable actual types
`type T (<>) is limited private;`	Any type at all. The actual type can be limited or not, indefinite or definite, but the *generic* treats it as limited and indefinite, i.e. does not assume that assignment is available for the type.
`type T (<>) is private;`	Any nonlimited type: the generic knows that it is possible to assign to variables of this type, but it is not possible to declare objects of this type without initial value.
`type T is private;`	Any nonlimited definite type: the generic knows that it is possible to assign to variables of this type and to declare objects without initial value.
`type T (<>) is abstract tagged limited private;`	Any tagged type, abstract or concrete, 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 concrete.
`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 concrete, 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 mod <>;`	Any modular type
`type T is delta <>;`	Any (non-decimal) fixed point type
`type T is delta <> digits <>;`	Any decimal fixed point type
`type T is digits <>;`	Any floating 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 category 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++.

It is possible to further restrict the set of acceptable actual types like so:

Generic formal type	Acceptable actual types
`type T (<>) is...`	Definite or indefinite types (loosely speaking: types with or without discriminants, but other forms of indefiniteness exist)
`type T (D : DT) is...`	Types with a discriminant of type DT (it is possible to specify several discriminants, too)
`type T is...`	Definite types (loosely speaking types without a discriminant or with a discriminant with default value)

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 : Matrices.Matrix_T := Matrices.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 (https://sourceforge.net/p/wikibook-ada/code/HEAD/tree/trunk/demos/Source/Algorithms/binary_search.adb), plain text (https://sourceforge.net/p/wikibook-ada/code/HEAD/tree/trunk/demos/Source/Algorithms/binary_search.adb?format=raw), download page (https://sourceforge.net/project/showfiles.php?group_id=124904), browse all (https://wikibook-ada.sourceforge.net/samples))

```
  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;
```

You can specify one default parameters, none or only some. Defaults are indicated with a box (" => <> "), and you can use " others => <>") to mean "use defaults for all parameters not mentioned". 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 sees everything 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 (http://www.ada-auth.org/standards/12rm/html/RM-12.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-12.html)]

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 dependent tasks have terminated. The environment task is the master of all other tasks; it terminates only when all other tasks 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 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 poor description.)

There is, however, a big difference between a procedure and an entry. A procedure has exactly one body that is executed when called. There is no such relation between an entry and a corresponding accept statement. An entry may have more than one accept statement, and the code executed may be different each time. In fact, there even need not be an accept statement at all. (Calling such an entry leads to deadlock of the caller if not timed, of course.)

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, based on the well-known computer science concept of a *monitor*. 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.)

Similar to a task entry which optionally has a *guard*, a protected entry must have a *barrier* to control admission. This provides automatic signaling, and ensures that when a protected entry call is accepted, its barrier condition is True, so that a barrier provides a reliable precondition for the entry body. A barrier can statically be true, then the entry is always open.

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 barriers, 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.

```
subtype 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 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 the rendezvous is not immediately achieved. This uses the select statement notation with an **else** part. Thus the constructs

```
   select
     Callee.Rendezvous;
   else
     Do_something_else;
   end select;
```

and

```
   select
     Callee.Rendezvous;
   or
     delay 0.0;
     Do_something_else;
   end select;
```

seem to be conceptually equivalent. However, the attempt to start the rendezvous may take some time, especially if the callee is on another processor, so the *delay 0.0;* may expire although the callee would be able to accept the rendezvous, whereas the *else* construct is safe.

Requeue statements

A requeue statement allows an accept statement or entry body to be completed while redirecting it to a different or the same entry queue, even to one of another task. The called entry has to share the same parameter list or be parameter-less. The caller of the original entry is not aware of the requeue and the entry call continues although now to possibly another entry of another task.

The requeue statement should normally be used to quickly check some precondition for the work proper. If these are fulfilled, the work proper is delegated to another task, hence the caller should nearly immediately be requeued.

Thus requeuing may have an effect on timed entry calls. To be a bit more specific, say the timed entry call is to T1.E1, the requeue within T1.E1 to T2.E2:

```
task body T1 is
  ...
  accept E1 do
    ...  --  Here quick check of preconditions.
    requeue T2.E2;  --  delegation
  end E1;
  ...
end T1;
```

Let Delta_T be the timeout of the timed entry call to T1.E1. There are now several possibilities:

1. Delta_T expires before T1.E1 is accepted.

    The entry call is aborted, i.e. taken out of the queue.

2. Delta_T expires after T1.E1 is accepted.

    T1.E1 has finished (its check of preconditions) and T2.E2 is to be accepted.
    For the caller, who is unaware of the requeue, the entry call is still executing; it is completed only with the completion T2.E2.

Thus, although the original entry call may be postponed for a long time while T2.E2 is waiting to be accepted, the call is executing from the caller's point of view.

To avoid this behaviour, a call may be *requeued with abort*. This changes case 2 above:

2.a The call is requeued to T2.E2 before Delta_T expires.

    2.a.1. T2.E2 is accepted before expiration, the call continues until T2.E2 is completed.

    2.a.2. Delta_T expires before T2.E2 is accepted: The entry call is aborted, i.e. taken out of the queue of T2.E2.

2.b The call is requeued to T2.E2 after the expiration of Delta_T.

    2.b.1. T2.E2 is immediately available (i.e. there is no requeue), T2.E2 continues to completion.

    2.b.2. T2.E2 is queued: The entry call is aborted, i.e. taken out of the queue of T2.E2.

In short, for a requeue with abort, the entry call to T1.E1 is completed in cases 1, 2.a.1 and 2.b.1; it is aborted in 2.a.2 and 2.b.2.

So what is the difference between these three entries?

```
accept E1 do
   ...  --  Here quick check of preconditions.
   requeue T2.E2 with abort;  --  delegation
end E1;

accept E2 do
   ...  --  Here quick check of preconditions.
   T2.E2;  --  delegation
end E2;
```

```
   accept E3 do
     ...  --  Here quick check of preconditions.
   end E3;
   T2.E2;  --  delegation
```

E1 has just been discussed. After the requeue, its enclosing task is free for other work, while the caller is still suspended until its call is completed or aborted.

E2 also delegates, however via an entry call. Thus E2 completes only with the completion of T2.E2.

E3 first frees the caller, then delegates to T2.E2, i.e. the entry call is completed with E3.

Scheduling

FIFO, priority, priority inversion avoidance, ... to be completed.

Interfaces and polymorphism

This language feature is only available from Ada 2005 on.

Tasks 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;
```

To allow delegation necessary to the polymorphism, the interface **Printable** shall be defined in its own package. It is then possible to define different **task type** implementing the **Printable** interface and use these implemetations polymorphically:

```
with printable_package; use printable_package;  -- This package contains the definition of Printable
procedure Printer is
   task type Print_Red  is new Printable with end;
   task type Print_Blue is new Printable with end;

   task body Print_Red is
   begin
      Ada.Text_IO.Put_Line ("Printing in Red");
   end Print_Red;

   task body Print_Blue is
   begin
      Ada.Text_IO.Put_Line ("Printing in Blue");
   end Print_Blue;

   printer_task : access Printable'Class;
begin
   printer_task := new Print_Red;
   printer_task := new Print_Blue;  -- Beware, this leaks memory. Example only.
end Printer;
```

This feature is also called **synchronized interfaces**.

Restrictions and Profiles

Ada tasking has too many features for some applications. Therefore there exist restrictions and profiles for certain, mostly safety or security critical applications. Restrictions and profiles are defined via pragmas. A restriction forbids the use of certain features, for instance the restriction No_Abort_Statements forbids the use of the abort statement. A profile (do not confuse with parameter profiles for subprograms) combines a set of restrictions.

See 13.12: Pragma Restrictions and Pragma Profile (http://www.ada-auth.org/standards/12rm/html/RM-13-12.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-13-12.html)]

See also

Wikibook

- Ada Programming

- Ada Programming/Libraries/Ada.Storage IO
- Ada Programming/Libraries/Ada.Task_Identification
- Ada Programming/Libraries/Ada.Task_Attributes

Ada Reference Manual

Ada 95

- Section 9: Tasks and Synchronization (http://www.adaic.com/standards/95lrm/html/RM-9.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-9.html)]

Ada 2005

- 3.9.4: Interface Types (http://www.adaic.com/standards/05rm/html/RM-3-9-4.html) [Annotated (http://www.adaic.com/standards/05aarm/html/AA-3-9-4.html)]
- Section 9: Tasks and Synchronization (http://www.adaic.com/standards/05rm/html/RM-9.html) [Annotated (http://www.adaic.com/standards/05aarm/html/AA-9.html)]

Ada Quality and Style Guide

- **Chapter 4: Program Structure**
  - 4.1.9 Tasks (http://www.adaic.org/docs/95style/html/sec_4/4-1-9.html)
  - 4.1.10 Protected Types (http://www.adaic.org/docs/95style/html/sec_4/4-1-10.html)
- Chapter 6: Concurrency (http://www.adaic.org/docs/95style/html/sec_6/toc.html)

Object Orientation

Object orientation in Ada

Object oriented programming consists in building the software in terms of "objects". An "object" contains data and has a behavior. The data, normally, consists in constants and variables as seen in the rest of this book but could also, conceivably, reside outside the program entirely, i.e. on disk or on the network. The behavior consists in subprograms that operate on the data. What makes Object Orientation unique, compared to procedural programming, is not a single feature but the combination of several features:

- *encapsulation*, i.e. the ability to separate the implementation of an object from its interface; this in turn separates "clients" of the object, who can only use the object in certain predefined ways, from the internals of the object, which have no knowledge of the outside clients.
- *inheritance*, the ability for one type of objects to inherit the data and behavior (subprograms) of another, without necessarily needing to break encapsulation;
- *type extension*, the ability for an object to add new data components and new subprograms on top of the inherited ones and to *replace* some inherited subprograms with its own versions; this is called *overriding*.
- *polymorphism*, the ability for a "client" to use the services of an object without knowing the exact type of the object, i.e. in an abstract way. Indeed at run time, the actual objects can have different types from one invocation to the next.

It is possible to do object-oriented programming in any language, even assembly. However, type extension and polymorphism are very difficult to get right without language support.

In Ada, each of these concepts has a matching construct; this is why Ada supports object-oriented programming directly.

- Packages provide encapsulation;
- Derived types provide inheritance;
- Record extensions, described below, provide for type extension;
- Class-wide types, also described below, provide for polymorphism.

Ada has had encapsulation and derived types since the first version (MIL-STD-1815 in 1980), which led some to qualify the language as "object-oriented" in a very narrow sense. Record extensions and class-wide types were added in Ada 95. Ada 2005 further adds interfaces. The rest of this chapter covers these aspects.

The simplest object: the Singleton

```
package Directory is
  function Present (Name_Pattern: String) return Boolean;
  generic
     with procedure Visit (Full_Name, Phone_Number, Address: String;
                           Stop: out Boolean);
```

```
   procedure Iterate (Name_Pattern: String);
end Directory;
```

The Directory is an object consisting of data (the telephone numbers and addresses, presumably held in an external file or database) and behavior (it can look an entry up and traverse all the entries matching a Name_Pattern, calling Visit on each).

A simple package provides for encapsulation (the inner workings of the directory are hidden) and a pair of subprograms provide the behavior.

This pattern is appropriate when only one object of a certain type must exist; there is, therefore, no need for type extension or polymorphism.

Primitive operations

In Ada, methods are usually referred to by the technical term *primitive subprograms of a tagged type* or the equivalent term *primitive operations of a tagged type*. The primitive operations of a type are those that are always available wherever the type is used. For the tagged types that are used in object oriented programming, they also can be inherited by and overridden by derived types, and can be dynamically dispatched.

Primitive operations of a type need to be declared immediately within the same package as the type (not within a nested package nor a child package). For tagged types, new primitive operations and overrides of inherited primitive operations are further required to be declared before the *freezing point* of the type. Any subprograms declared after the freezing point will not be considered primitive, and therefore cannot be inherited and are not dynamically dispatched. Freezing points are discussed in more detail below, but the simple practice of declaring all primitive operations immediately following the initial type declaration will ensure that those subprograms are indeed recognized as primitive.

Primitive operations of type `T` need to have at least one parameter of type `T` or of type `access T`. While most object-oriented languages automatically provide a `this` or `self` pointer, Ada requires that you explicitly declare a formal parameter to receive the current object. That typically will be the first parameter in the list, which enables the `object.subprogram` call syntax (available since Ada 2005), but it may be at any parameter position. Tagged types are always passed by reference; the parameter passing method has nothing to do with the parameter modes `in` and `out`, which describe the dataflow. The parameter passing method is identical for `T` and `access T`.

For tagged types, no other directly dispatchable types can be used in the parameter list because Ada doesn't offer multiple dispatching. The following example is illegal.

```
package P is
   type A is tagged private;
   type B is tagged private;
   procedure Proc (This: B; That: A); -- illegal: can't dispatch on both A and B
end P;
```

When additional dispatchable objects need to be passed in, the parameter list should declare them using their class-wide types, `T'Class`. For example:

```
package P is
   type A is tagged private;
   type B is tagged private;
   procedure Proc (This: B; That: A'Class); -- dispatching only on B
end P;
```

Note, however, that this does not limit the number of parameters of the same tagged type. For example, the following definition is legal.

```
package P is
   type A is tagged private;
   procedure Proc (This, That: A); -- dispatching only on A
end P;
```

Primitive operations of tagged types are dispatching operations. Whether a call to such a primitive operation is in effect dispatching or statically bound, depends on the context (see below). Note that in a dispatching call both actual parameters of the last example must have the same tag (i.e. the same type); Constraint_Error will be called if the tag check fails.

Derived types

Type derivation has been part of Ada since the very start.

```
package P is
  type T is private;
  function Create (Data: Boolean) return T;  -- primitive
  procedure Work (Object : in out T);        -- primitive
  procedure Work (Pointer: access T);        -- primitive
  type Acc_T is access T;
  procedure Proc (Pointer: Acc_T);           -- not primitive
private
  type T is record
    Data: Boolean;
  end record;
end P;
```

The above example creates a type T that contains data (here just a Boolean but it could be anything) and behavior consisting of some subprograms. It also demonstrates encapsulation by placing the details of the type T in the private part of the package.

The primitive operations of T are the function Create, the overloaded procedures Work, and the predefined "=" operator; Proc is not primitive, since it has an *access type* on T as parameter — don't confuse this with an *access parameter*, as used in the second procedure Work. When deriving from T, the primitive operations are inherited.

```
with P;
package Q is
  type Derived is new P.T;
end Q;
```

The type Q.Derived has the same data *and the same behavior* as P.T; it inherits both the data *and the subprograms*. Thus it is possible to write:

```
with Q;
procedure Main is
  Object: Q.Derived := Q.Create (Data => False);
begin
  Q.Work (Object);
end Main;
```

Inherited operations may be overridden and new operations added, but the rules (Ada 83) unfortunaltely are somewhat different from the rules for tagged types (Ada 95).

Admittedly, the reasons for writing this may seem obscure. The purpose of this kind of code is to have objects of types P.T and Q.Derived, which are not compatible:

```
Ob1: P.T;
Ob2: Q.Derived;
```

```
Ob1 := Ob2;               -- illegal
Ob1 := P.T (Ob2);         -- but convertible
Ob2 := Q.Derived (Ob1);   -- in both directions
```

This feature is not used very often (it's used e.g. for declaring types reflecting physical dimensions) but I present it here to introduce the next step: type extension.

Type extensions

Type extensions are an Ada 95 amendment.

A tagged type provides support for dynamic polymorphism and type extension. A tagged type bears a hidden tag that identifies the type at run-time. Apart from the tag, a tagged record is like any other record, so it can contain arbitrary data.

```
package Person is
   type Object is tagged
     record
         Name   : String (1 .. 10);
         Gender : Gender_Type;
     end record;
   procedure Put (O : Object);
end Person;
```

As you can see, a `Person.Object` is an *object* in the sense that it has data and behavior (the procedure `Put`). However, this object does not hide its data; any program unit that has a `with Person` clause can read and write the data in a Person.Object directly. This breaks encapsulation and also illustrates that Ada completely separates the concepts of *encapsulation* and *type*. Here is a version of Person.Object that encapsulates its data:

```
package Person is
   type Object is tagged private;
   procedure Put (O : Object);
private
   type Object is tagged
     record
         Name   : String (1 .. 10);
         Gender : Gender_Type;
     end record;
end Person;
```

Because the type Person.Object is tagged, it is possible to create a record extension, which is a derived type with additional data.

```
with Person;
package Programmer is
   type Object is new Person.Object with private;
private
   type Object is new Person.Object with
     record
        Skilled_In : Language_List;
     end record;
end Programmer;
```

The type `Programmer.Object` inherits the data and behavior, i.e. the type's primitive operations, from `Person.Object`; it is thus possible to write:

```
with Programmer;
procedure Main is
   Me : Programmer.Object;
begin
   Programmer.Put (Me);
   Me.Put; -- equivalent to the above, Ada 2005 only
end Main;
```

So the declaration of the type `Programmer.Object`, as a record extension of `Person.Object`, implicitly declares a `procedure Put` that applies to a `Programmer.Object`.

Like in the case of untagged types, objects of type Person and Programmer are convertible. However, where untagged objects are convertible in either direction, conversion of tagged types only works in the direction to the root. (Conversion away from the root would have to add components out of the blue.) Such a conversion is called a *view conversion*, because components are not lost, they only become invisible.

Extension aggregates have to be used if you go away from the root.

Overriding

Now that we have introduced tagged types, record extensions and primitive operations, it becomes possible to understand overriding. In the examples above, we introduced a type `Person.Object` with a primitive operation called `Put`. Here is the body of the package:

```
with Ada.Text_IO;
package body Person is
   procedure Put (O : Object) is
   begin
      Ada.Text_IO.Put (O.Name);
      Ada.Text_IO.Put (" is a ");
      Ada.Text_IO.Put_Line (Gender_Type'Image (O.Gender));
   end Put;
end Person;
```

As you can see, this simple operation prints both data components of the record type to standard output. Now, remember that the record extension `Programmer.Object` has an additional data member. If we write:

```
with Programmer;
procedure Main is
   Me : Programmer.Object;
begin
   Programmer.Put (Me);
```

```
   Me.Put; -- equivalent to the above, Ada 2005 only
end Main;
```

then the program will call the inherited primitive operation `Put`, which will print the name and gender *but not the additional data*. In order to provide this extra behavior, we must *override* the inherited procedure `Put` like this:

```
with Person;
package Programmer is
   type Object is new Person.Object with private;
   overriding -- Optional keyword, new in Ada 2005
   procedure Put (O : Object);
private
   type Object is new Person.Object with
     record
        Skilled_In : Language_List;
     end record;
end Programmer;
```

```
package body Programmer is
   procedure Put (O : Object) is
   begin
      Person.Put (Person.Object (O)); -- view conversion to the ancestor type
      Put (O.Skilled_In); -- presumably declared in the same package as Language_List
   end Put;
end Programmer;
```

`Programmer.Put` *overrides* `Person.Put`; in other words it *replaces* it completely. Since the intent is to extend the behavior rather than replace it, `Programmer.Put` calls `Person.Put` as part of its behavior. It does this by converting its parameter from the type `Programmer.Object` to its ancestor type `Person.Object`. This construct is a *view conversion*; contrary to a normal type conversion, it does *not* create a new object and does *not* incur any run-time cost (and indeed, if the operand of such *view conversion* was actually a variable, the result can be used when an out parameter is required (eg. procedure call)). Of course, it is optional that an overriding operation call its ancestor; there are cases where the intent is indeed to replace, not extend, the inherited behavior.

(Note that also for untagged types, overriding of inherited operations is possible. The reason why it's discussed here is that derivation of untagged types is done rather seldom.)

Polymorphism, class-wide programming and dynamic dispatching

The full power of object orientation is realized by polymorphism, class-wide programming and dynamic dispatching, which are different words for the same, single concept. To explain this concept, let us extend the example from the previous sections, where we declared a base tagged type `Person.Object` with a primitive operation `Put` and a record extension `Programmer.Object` with additional data and an overriding primitive operation `Put`.

Now, let us imagine a collection of persons. In the collection, some of the persons are programmers. We want to traverse the collection and call `Put` on each person. When the person under consideration is a programmer, we want to call `Programmer.Put`; when the person is not a programmer, we want to call `Person.Put`. This, in essence, is polymorphism, class-wide programming and dynamic dispatching.

With Ada's strong typing, ordinary calls cannot be dynamically dispatched; a call to an operation on a declared type must always be statically bound to go to the operation defined for that specific type. Dynamic dispatching (known as simply *dispatching* in Ada parlance) is provided through separate *class-wide types* that are polymorphic. Each tagged type, such as `Person.Object`, has a corresponding *class of types* which is the set of types comprising the type `Person.Object` itself and all types that extend `Person.Object`. In our example, this class consists of two types:

- `Person.Object`
- `Programmer.Object`

Ada 95 defines the `Person.Object'Class` attribute to denote the corresponding class-wide type. In other words:

```
declare
   Someone : Person.Object'Class := ...; -- to be expanded later
begin
   Someone.Put; -- dynamic dispatching
end;
```

The declaration of Someone denotes an object that may be of *either* type, `Person.Object` or `Programmer.Object`. Consequently, the call to the primitive operation `Put` dispatches dynamically to either `Person.Put` or `Programmer.Put`.

The only problem is that, since we don't know whether Someone is a programmer or not, we don't know how many data components Someone has, either, and therefore we don't know how many bytes Someone takes in memory. For this reason, the class-wide type `Person.Object'Class` is *indefinite*. It is impossible to declare an object of this type without giving some constraint. It is, however, possible to:

- declare an object of a class-wide with an initial value (as above). The object is then constrained by its initial value.
- declare an *access value* to such an object (because the access value has a known size);
- pass objects of a class-wide type as parameters to subprograms

- assign an object of a specific type (in particular, the result of a function call) to a variable of a class-wide type.

With this knowledge, we can now build a polymorphic collection of persons; in this example we will quite simply create an array of access values to persons:

```
with Person;
procedure Main is
   type Person_Access is access Person.Object'Class;
   type Array_Of_Persons  is array (Positive range <>) of Person_Access;

   function Read_From_Disk return Array_Of_Persons is separate;

   Everyone : constant Array_Of_Persons := Read_From_Disk;
begin -- Main
   for K in Everyone'Range loop
      Everyone (K).all.Put; -- dereference followed by dynamic dispatching
   end loop;
end Main;
```

The above procedure achieves our desired goal: it traverses the array of Persons and calls the procedure `Put` that is appropriate for each person.

Advanced topic: How dynamic dispatching works

You don't need to know how dynamic dispatching works in order to use it effectively but, in case you are curious, here is an explanation.

The first component of each object in memory is the *tag*; this is why objects are of a *tagged* type rather than plain records. The tag really is an access value to a table; there is one table for each specific type. The table contains access values to each primitive operation of the type. In our example, since there are two types `Person.Object` and `Programmer.Object`, there are two tables, each containing a single access value. The table for `Person.Object` contains an access value to `Person.Put` and the table for `Programmer.Object` contains an access value to `Programmer.Put`. When you compile your program, the compiler constructs both tables and places them in the program executable code.

Each time the program creates a new object of a specific type, it automatically sets its tag to point to the appropriate table.

Each time the program performs a *dispatching call* of a primitive operation, the compiler inserts object code that:

- dereferences the tag to find the table of primitive operations for the specific type of the object at hand
- dereferences the access value to the primitive operation
- calls the primitive operation.

Conversely, when the program performs a call where the parameter is a view conversion to an ancestor type, the compiler performs these two dereferences at compile time rather than run time: such a call is *statically bound*; the compiler emits code that directly calls the primitive operation of the ancestor type specified in the view conversion.

Redispatching

Dispatching is controlled by the (hidden) tag of the object. So what happens when a primitive operation `Op1` calls another primitive operation `Op2` on the same object?

```
type Root is tagged private;
procedure Op1 (This: Root);
procedure Op2 (This: Root);

type Derived is new Root with private;
-- Derived inherits Op1
overriding procedure Op2 (This: Derived);

procedure Op1 (This: Root) is
begin
  ...
  Op2 (This);               -- not redispatching
  Op2 (Root'Class (This));  -- redispatching
  This.Op2;                 -- not redispatching (new syntax since Ada 2005)
  (Root'Class (This)).Op2;  -- redispatching (new syntax since Ada 2005)
  ...
end Op1;

D: Derived;
C: Root'Class := D;

Op1 (D);  -- statically bound call
Op1 (C);  -- dispatching call
D.Op1;    -- statically bound call (new syntax since Ada 2005)
C.Op1;    -- dispatching call (new syntax since Ada 2005)
```

In this fragment, `Op1` is not overridden, whereas `Op2` is overridden. The body of `Op1` calls `Op2`, so which `Op2` will be called if `Op1` is called for an object of type `Derived`?

The basic rules of dispatching still apply. Calls to `Op2` will be dispatched when called using an object of a class-wide type.

The formal parameter lists for the operations specify the type of `This` to be a specific type, not class-wide. In fact, that parameter *must* be a specific type so that the operation will be dispatched for objects of that type, and to allow the operation's code to access any additional data items associated with that type. If you want redispatching, you must state that explicitly by converting the parameter of the specific type to the class-

wide type again. (Remember: view conversions never lose components, they just hide them. A conversion to a class-wide type can unhide them again.) The first call `Op1 (D)` (statically bound, i.e., not dispatching) executes the inherited `Op1` — and within `Op1`, the first call to `Op2` is also statically bound (there is no redispatching) because parameter `This` is a view conversion to specific type `Root`. However, the second call is dispatching because the parameter `This` is converted to the class-wide type. That call dispatches to the overriding `Op2`.

Because the conventional `This.Op2` call is *not* dispatching, the call will be to `Root.Op2` even though the object itself is of type `Derived` and the `Op2` operation is overridden. *This is very different from how other OO languages behave*. In other OO languages, a method is either dispatching or not. In Ada, an operation is either *available* for dispatching or not. Whether or not dispatching is actually *used* for a given call depends on the way that the object's type is specified at that call point. For programmers accustomed to other OO languages, it can come as quite a surprise that calls from a dispatchable operation to other operations on the same object are, by default, *not* (dynamically) dispatched.

The default of not redispatching is not an issue if all of the operations have been overridden, because they all will be operating on the expected type of object. However, it has ramifications when writing code for types that might be extended by another type sometime in the future. It's possible that the new type will not work as intended if it doesn't override all of the primitive operations that call other primitive operations. The safest policy is to use a class-wide conversion of the object to force dispatching of calls. One way to accomplish that is to define a class-wide constant in each dispatched method:

```
procedure Op2 (This: Derived) is
  This_Class: constant Root'Class := This;
begin
```

`This` is needed to access data items and to make any non-dispatching calls. `This_Class` is needed to make dispatching calls.

Less commonly encountered and perhaps less surprising, calls from a non-dispatchable (class-wide) routine for a tagged type to other routines on the same object are, by default, dispatched:

```
type Root is tagged private;
procedure Op1 (This: Root'Class);
procedure Op2 (This: Root);

type Derived is new Root with private;
-- Derived does not inherit Op1, rather Op1 is applicable to Derived.
overriding procedure Op2 (This: Derived);

procedure Op1 (This: Root'Class) is
begin
  ...
  Op2 (This);               -- dispatching
  Op2 (Root (This));        -- static call
  This.Op2;                 -- dispatching (new syntax since Ada 2005)
```

```
   (Root (This)).Op2;        -- static call (new syntax since Ada 2005)
   ...
end Op1;

D: Derived;
C: Root'Class := D;

Op1 (D);  -- static call
Op1 (C);  -- static call
D.Op1;    -- static call (new syntax since Ada 2005)
C.Op1;    -- static call (new syntax since Ada 2005)
```

Note that calls on `Op1` are always static, since `Op1` is *not* inherited. Its parameter type is class-wide, so the operation is *applicable* to all types derived from Root. (Op2 has an entry for each type derived from `Root` in the dispatch table. There is no such dispatch table for `Op1`; rather there is only one such operation for all types in the class.)

Normal calls *from* `Op1` are dispatched because the declared type of `This` is class-wide. The default to dispatching usually isn't bothersome because class-wide operations are typically used to perform a script involving calls to one or more dispatched operations.

Run-time type identification

Run-time type identification allows the program to (indirectly or directly) query the tag of an object at run time to determine which type the object belongs to. This feature, obviously, makes sense only in the context of polymorphism and dynamic dispatching, so works only on tagged types.

You can determine whether an object belongs to a certain class of types, or to a specific type, by means of the membership test **`in`**, like this:

```
type Base    is tagged private;
type Derived is new Base    with private;
type Leaf    is new Derived with private;

...
procedure Explicit_Dispatch (This : in Base'Class) is
begin
   if This in Leaf then ... end if;
   if This in Derived'Class then ... end if;
end Explicit_Dispatch;
```

Thanks to the strong typing rules of Ada, run-time type identification is in fact rarely needed; the distinction between class-wide and specific types usually allows the programmer to ensure objects are of the appropriate type without resorting to this feature.

Additionally, the reference manual defines `package Ada.Tags` (RM 3.9(6/2)), attribute `'Tag` (RM 3.9(16,18)), and `function Ada.Tags.Generic_Dispatching_Constructor` (RM 3.9(18.2/2)), which enable direct manipulation with tags.

Creating Objects

The Language Reference Manual's section on 3.3: Objects and Named Numbers (http://www.ada-auth.org/standards/12rm/html/RM-3-3.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-3-3.html)] states when an object is created, and destroyed again. This subsection illustrates how objects are created.

The LRM section 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 example, assume a typical hierarchy of object oriented types: a top-level type `Person`, a `Programmer` type derived from `Person`, and possibly more kinds of persons. Each person has a name; assume `Person` objects to have a `Name` component. Likewise, each `Person` has a `Gender` component. The `Programmer` type inherits the components and the operations of the `Person` type, so `Programmer` objects have a `Name` and a `Gender` component, too. `Programmer` objects may have additional components specific to programmers.

Objects of a tagged type are created the same way as objects of any type. The second LRM sentence says, 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. One way of giving initial values to the object's components is to assign an aggregate.

```
declare
   P: Person := (Name => "Scorsese", Gender => Male);
begin
   Text_IO.Put_Line("The name is " & P.Name);
end;
```

The parenthesized expression after := is called an *aggregate* (4.3: Aggregates (http://www.ada-auth.org/standards/12rm/html/RM-4-3.html) [Annotated (http://www.ada-auth.org/standards/12aarm/html/AA-4-3.html)]).

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. Therefore, instead of using an aggregate of initial values, we might call a function returning an object.

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. For instance, the aggregate above will most probably raise the exception Constraint_Error depending on the name string supplied; the function can mangle the name so that it matches the declaration of the component.) 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 an object which can be used for initialization. Since the `Person` type is **private** we can no longer refer to the `Name` component of `P`. But there is a corresponding function `Name` declared with type `Person` making it a socalled primitive operation. (The component and the function in this example are both named `Name` However, we can choose a different name for either if we want.)

```
declare
   P: Person := Make (Name => "Orwell", Sex => Male);
begin
   Text_IO.Put_Line("The name is " & Name(P));
end;
```

Objects can be copied into another. The target object is first destroyed. Then the component values of the source object are assigned to the corresponding components of the target object. In the following example, the default initialized `P` gets a copy of one of the objects created by the `Make` calls.

```
declare
   P: Person;
begin
   if 2001 > 1984 then
      P := Make (Name => "Kubrick", Sex => Male);
   else
      P := Make (Name => "Orwell", 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`. There is no polymorphism yet, and likewise initialization does not yet mention inheritance. Before dealing with `Programmer` objects and their initialization a few words about class-wide types are in order.

More details on primitive operations

Remember what we said before about "Primitive Operations". Primitive operations are:

- subprograms taking a parameter of the tagged type;
- functions returning an object of the tagged type;
- subprograms taking a parameter of an *anonymous access type* to the tagged type;
- In Ada 2005 only, functions returning an *anonymous access type* to the tagged type;

Additionally, primitive operations must be declared before the type is *frozen* (the concept of freezing will be explained later):

Examples:

```
package X is
   type Object is tagged null record;

   procedure Primitive_1 (This : in     Object);
   procedure Primitive_2 (That :    out Object);
   procedure Primitive_3 (Me   : in out Object);
   procedure Primitive_4 (Them : access Object);
   function  Primitive_5 return Object;
```

```
   function  Primitive_6 (Everyone : Boolean) return access Object;
end X;
```

All of these subprograms are primitive operations.

A primitive operation can also take parameters of the same or other types; also, the controlling operand does not have to be the first parameter:

```
package X is
   type Object is tagged null record;

   procedure Primitive_1 (This : in Object; Number : in Integer);
   procedure Primitive_2 (You  : in Boolean; That : out Object);
   procedure Primitive_3 (Me, Her : in out Object);
end X;
```

The definition of primitive operations specifically excludes named access types and class-wide types as well as operations not defined immediately in the same declarative region. Counter-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 Not_Primitive_1 (This : in     Object'Class);
   procedure Not_Primitive_2 (This : in out Object_Access);
   procedure Not_Primitive_3 (This :    out Object_Class_Access);
   function  Not_Primitive_4 return Object'Class;

   package Inner is
       procedure Not_Primitive_5 (This : in Object);
   end Inner;
end X;
```

Advanced topic: Freezing rules

Freezing rules (ARM 13.14 (http://www.adaic.com/standards/05rm/html/RM-13-14.html)) are perhaps the most complex part of the Ada language definition; this is because the standard tries to describe freezing as unambiguously as possible. Also, that part of the language definition deals with freezing of all entities, including complicated situations like generics and objects reached by dereferencing access values. You can, however, get an intuitive understanding of freezing of tagged types if you understand how dynamic dispatching works. In that section, we saw

that the compiler emits a table of primitive operations for each tagged type. The point in the program text where this happens is the point where the tagged type is *frozen*, i.e. the point where the table becomes complete. After the type is frozen, no more primitive operations can be added to it.

This point is the earliest of:

- the end of the package spec where the tagged type is declared
- the appearance of the first type derived from the tagged type

Example:

```
package X is

  type Object is tagged null record;
  procedure Primitive_1 (This: in Object);

  -- this declaration freezes Object
  type Derived is new Object with null record;

  -- illegal: declared after Object is frozen
  procedure Primitive_2 (This: in Object);

end X;
```

Intuitively: at the point where Derived is declared, the compiler starts a new table of primitive operations for the derived type. This new table, initially, is equal to the table of the primitive operations of the parent type, `Object`. Hence, `Object` must freeze.

- the declaration of a variable of the tagged type

Example:

```
package X is

  type Object is tagged null record;
  procedure Primitive_1 (This: in Object);

  V: Object;  -- this declaration freezes Object

  -- illegal: Primitive operation declared after Object is frozen
  procedure Primitive_2 (This: in Object);

end X;
```

Intuitively: after the declaration of V, it is possible to call any of the primitive operations of the type on V. Therefore, the list of primitive operations must be known and complete, i.e. frozen.

- The completion (*not* the declaration, if any) of a constant of the tagged type:

```
package X is

  type Object is tagged null record;
  procedure Primitive_1 (This: in Object);

  -- this declaration does NOT freeze Object
  Deferred_Constant: constant Object;

  procedure Primitive_2 (This : in Object); -- OK

private

  -- only the completion freezes Object
  Deferred_Constant: constant Object := (null record);

  -- illegal: declared after Object is frozen
  procedure Primitive_3 (This: in Object);

 end X;
```

New features of Ada 2005

This language feature is only available from Ada 2005 on.

Ada 2005 adds overriding indicators, allows anonymous access types in more places and offers the object.method notation.

Overriding indicators

The new keyword **overriding** can be used to indicate whether an operation overrides an inherited subprogram or not. Its use is optional because of upward-compatibility with Ada 95. For example:

```
package X is
    type Object is tagged null record;

   function  Primitive return access Object; -- new in Ada 2005

   type Derived_Object is new Object with null record;
```

```
   not overriding -- new optional keywords in Ada 2005
   procedure Primitive (This : in Derived_Object); -- new primitive operation

   overriding
   function  Primitive return access 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 Primitive_X (This : in Object) is abstract;

overriding
function  Primitive_Y return Object renames Some_Other_Subprogram;

not overriding
procedure Primitive_Z (This : out Object)
      is new Generic_Procedure (Element => Integer);
```

Object.Method notation

We have already seen this notation:

```
package X is
   type Object is tagged null record;

   procedure Primitive (This: in Object; That: in Boolean);
end X;
```

```
with X;
procedure Main is
   Obj : X.Object;
begin
   Obj.Primitive (That => True); -- Ada 2005 object.method notation
end Main;
```

This notation is only available for primitive operations where the controlling parameter is the *first* parameter.

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.

Multiple Inheritance via Interfaces

This language feature is only available from Ada 2005 on.

Interfaces allow for a limited form of multiple inheritance (taken from Java). On a semantic level they are similar to an "abstract tagged null record" as they may have primitive operations but cannot hold any data and thus these operations cannot have a body, 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.

Such a type may implement a list of interfaces (called the *progenitors*), but can have only one *parent*. The parent may be a concrete type or also an interface.

```
type Derived is new Parent and Progenitor_1 and Progenitor_2 ... with ...;
```

Multiple Inheritance via Mix-in

Ada supports multiple inheritance of *interfaces* (see above), but only single inheritance of *implementation*. This means that a tagged type can *implement* multiple interfaces but can only *extend* a single ancestor tagged type.

This can be problematic if you want to add behavior to a type that already extends another type; for example, suppose you have

```
type Base is tagged private;
type Derived is new Base with private;
```

and you want to make `Derived` controlled, i.e. add the behavior that `Derived` controls its initialization, assignment and finalization. Alas you cannot write:

```
type Derived is new Base and Ada.Finalization.Controlled with private; -- illegal
```

since `Ada.Finalization` for historical reasons does not define interfaces `Controlled` and `Limited_Controlled`, but abstract types.

If your base type is not limited, there is no good solution for this; you have to go back to the root of the class and make it controlled. (The reason will become obvious presently.)

For limited types however, another solutions is the use of a mix-in:

```
type Base is tagged limited private;
type Derived;

type Controlled_Mix_In (Enclosing: access Derived) is
  new Ada.Finalization.Limited_Controlled with null record;

overriding procedure Initialize (This: in out Controlled_Mix_In);
overriding procedure Finalize   (This: in out Controlled_Mix_In);

type Derived is new Base with record
  Mix_In: Controlled_Mix_In (Enclosing => Derived'Access); -- special syntax here
  -- other components here...
end record;
```

This special kind of mix-in is an object with an access discriminant that references its enclosing object (also known as *Rosen trick*). In the declaration of the `Derived` type, we initialize this discriminant with a special syntax: `Derived'Access` really refers to an access value to the *current instance* of type `Derived`. Thus the access discriminant allows the mix-in to see its enclosing object and all its components; therefore it can initialize and finalize its enclosing object:

```
overriding procedure Initialize (This: in out Controlled_Mix_In) is
  Enclosing: Derived renames This.Enclosing.all;
begin
  -- initialize Enclosing...
end Initialize;
```

and similarly for `Finalize`.

The reason why this does not work for non-limited types is the self-referentiality via the discriminant. Imagine you have two variables of such a non-limited type and assign one to the other:

```
X := Y;
```

In an assignment statement, `Adjust` is called only *after* `Finalize` of the target `X` and so cannot provide the new value of the discriminant. Thus `X.Mixin_In.Enclosing` will inevitably reference `Y`.

Now let's further extend our hierarchy:

```
type Further is new Derived with null record;

overriding procedure Initialize (This: in out Further);
overriding procedure Finalize   (This: in out Further);
```

Oops, this does not work because there are no corresponding procedures for `Derived`, yet – so let's quickly add them.

```
type Base is tagged limited private;
type Derived;

type Controlled_Mix_In (Enclosing: access Derived) is
  new Ada.Finalization.Limited_Controlled with null record;

overriding procedure Initialize (This: in out Controlled_Mix_In);
overriding procedure Finalize   (This: in out Controlled_Mix_In);

type Derived is new Base with record
  Mix_In: Controlled_Mix_In (Enclosing => Derived'Access);  -- special syntax here
  -- other components here...
end record;

not overriding procedure Initialize (This: in out Derived);  -- sic, they are new
not overriding procedure Finalize   (This: in out Derived);

type Further is new Derived with null record;
```

```
overriding procedure Initialize (This: in out Further);
overriding procedure Finalize   (This: in out Further);
```

We have of course to write `not overriding` for the procedures on `Derived` because there is indeed nothing they could override. The bodies are

```
not overriding procedure Initialize (This: in out Derived) is
begin
  -- initialize Derived...
end Initialize;
```

```
overriding procedure Initialize (This: in out Controlled_Mix_In) is
  Enclosing: Derived renames This.Enclosing.all;
```

```
begin
  Initialize (Enclosing);
end Initialize;
```

To our dismay, we have to learn that `Initialize/Finalize` for objects of type `Further` will not be called, instead those for the parent `Derived`. Why?

```
declare
  X: Further;  -- Initialize (Derived (X)) is called here
begin
  null;
end;  -- Finalize (Derived (X)) is called here
```

The reason is that the mix-in defines the local object `Enclosing` to be of type `Derived` in the renames-statement above. To cure this, we have necessarily to use the dreaded redispatch (shown in different but equivalent notations):

```
overriding procedure Initialize (This: in out Controlled_Mix_In) is
  Enclosing: Derived renames This.Enclosing.all;
begin
  Initialize (Derived'Class (Enclosing));
end Initialize;
```

```
overriding procedure Finalize (This: in out Controlled_Mix_In) is
  Enclosing: Derived'Class renames Derived'Class (This.Enclosing.all);
begin
  Enclosing.Finalize;
end Finalize;
```

```
declare
  X: Further;  -- Initialize (X) is called here
begin
  null;
end;  -- Finalize (X) is called here
```

Alternatively (and presumably better still) is to write

```
type Controlled_Mix_In (Enclosing: access Derived'Class) is
  new Ada.Finalization.Limited_Controlled with null record;
```

Then we automatically get redispatch and can omit the type conversions on `Enclosing`.

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 unintuitive 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.

Object-Oriented Ada for C++ programmers

In C++, the construct

```
 struct C {
   virtual void v();
   void w();
   static void u();
 };
```

is strictly equivalent to the following in Ada:

```
package P is
  type C is tagged null record;
  procedure V (This : in out C);        -- primitive operation, will be inherited upon derivation
  procedure W (This : in out C'Class);  -- not primitive, will not be inherited upon derivation
  procedure U;
end P;
```

In C++, member functions implicitly take a parameter `this` which is of type `C*`. In Ada, all parameters are explicit. As a consequence, the fact that `u()` does *not* take a parameter is implicit in C++ but explicit in Ada.

In C++, `this` is a pointer. In Ada, the explicit `This` parameter does not have to be a pointer; all parameters of a tagged type are implicitly passed by reference anyway.

Static dispatching

In C++, function calls dispatch statically in the following cases:

- the target of the call is an object type
- the member function is non-virtual

For example:

```
C object;
object.v();
object.w();
```

both dispatch statically. In particular, the static dispatch for v() may be confusing; this is because object is neither a pointer nor a reference. Ada behaves exactly the same in this respect, except that Ada calls this *static binding* rather than *dispatching*:

```
declare
   Object : P.C;
begin
   Object.V; -- statically bound
   Object.W; -- statically bound
end;
```

Dynamic dispatching

In C++, a function call dispatches dynamically if the two following conditions are met simultaneously:

- the target of the call is a pointer or a reference
- the member function is virtual.

For example:

```
C* object;
object->v(); // dynamic dispatch
object->w(); // static, non-virtual member function
object->u(); // illegal: static member function
C::u(); // static dispatch
```

In Ada, a primitive subprogram call dispatches (dynamically) if and only if:

- the target object is of a class-wide type;

Note: In Ada vernacular, the term *dispatching* always means *dynamic*.

For example:

```
declare
   Object : P.C'Class := ...;
begin
   P.V (Object); -- dispatching
   P.W (Object); -- statically bound: not a primitive operation
   P.U; -- statically bound
end;
```

As can be seen *there is no need for access types or pointers* to do dispatching in Ada. In Ada, *tagged types are always passed by-reference to subprograms* without the need for explicit access values.

Also note that in C++, the class serves as:

- the unit of encapsulation (Ada uses packages and visibility for this)
- the type, like in Ada.

As a consequence, you call C::u() in C++ because u() is encapsulated in C, but P.U in Ada since U is encapsulated in the *package* P, not the *type* C.

Class-wide and specific types

The most confusing part for C++ programmers is the concept of a "class-wide type". To help you understand:

- pointers and references in C++ are really, implicitly, class-wide;
- object types in C++ are really specific;
- C++ provides no way to declare the equivalent of:

```
type C_Specific_Access is access C;
```

- C++ provides no way to declare the equivalent of:

```
type C_Specific_Access_One is access C;
type C_Specific_Access_Two is access C;
```

which, in Ada, are two different, *incompatible* types, possibly allocating their memory from different storage pools!

- In Ada, you do *not* need access values for dynamic dispatching.
- In Ada, you use access values for dynamic memory management (only) and class-wide types for dynamic dispatching (only).
- In C++, you use pointers and references both for dynamic memory management and for dynamic dispatching.
- In Ada, class-wide types are explicit (with `'Class`).
- In C++, class-wide types are implicit (with `*` or `&`).

Constructors

in C++, a special syntax declares a constructor:

```
 class C {
    C(/* optional parameters */); // constructor
 };
```

A constructor cannot be virtual. A class can have as many constructors, differentiated by their parameters, as necessary.

Ada does not have such constructors. Perhaps they were not deemed necessary since in Ada, any function that returns an object of the tagged type can serve as a kind of constructor. This is however not the same as a real constructor like the C++ one; this difference is most striking in cases of derivation trees (see Finalization below). The Ada constructor subprograms do not have to have a special name and there can be as many constructors as necessary; each function can take parameters as appropriate.

```
package P is
  type T is tagged private;
  function Make                 return T;  -- constructor
  function To_T (From: Integer) return T;  -- another constructor
  procedure Make (This: out T);            -- not a constructor
private
```

```
   ...
end P;
```

If an Ada constructor function is also a primitive operation (as in the example above), it becomes abstract upon derivation and has to be overridden if the derived type is not itself abstract. If you do not want this, declare such functions in a nested scope.

In C++, one idiom is the *copy constructor* and its cousin the *assignment operator*:

```
class C {
   C(const C& that); // copies "that" into "this"
   C& operator= (const C& right); // assigns "right" to "this", which is "left"
};
```

This copy constructor is invoked implicitly on initialization, e.g.

```
C a = b; // calls the copy constructor
C c;
a = c;   // calls the assignment operator
```

Ada provides a similar functionality by means of *controlled types*. A controlled type is one that extends the predefined type `Ada.Finalization.Controlled`:

```
with Ada.Finalization;
package P is
  type T is new Ada.Finalization.Controlled with private;
  function Make return T;  -- constructor
private
  type T is ... end record;
  overriding procedure Initialize (This: in out T);
  overriding procedure Adjust     (This: in out T); -- copy constructor
end P;
```

Note that Initialize is not a constructor; it resembles the C++ constructor in some way, but is also very different. Suppose you have a type T1 derived from T with an appropriate overriding of Initialize. A real constructor (like the C++ one) would automatically first construct the parent components (T), then the child components. In Ada, this is not automatic. In order to mimic this in Ada, we have to write:

```
procedure Initialize (This: in out T1) is
begin
  Initialize (T (This));  -- Don't forget this part!
```

```
   ...  -- handle the new components here
end Initialize;
```

The compiler inserts a call to Initialize after each object of type T is allocated when no initial value is given. It also inserts a call to Adjust after each assignment to the object. Thus, the declarations:

```
A: T;
B: T := X;
```

will:

- allocate memory for A
- call Initialize (A)
- allocate memory for B
- copy the contents of X to B
- call Adjust (B)

Initialize (B) will not be called because of the explicit initialization.

So, the equivalent of a copy constructor is an overriding of Adjust.

If you would like to provide this functionality to a type that extends another, non-controlled type, see "Multiple Inheritance".

Destructors

In C++, a destructor is a member function with only the implicit `this` parameter:

```
class C {
   virtual ~C(); // destructor
}
```

While a constructor *cannot* be virtual, a destructor *must* be virtual if the class is to be used with dynamic dispatch (has virtual methods or derives from a class with virtual methods). C++ classes do not use dynamic dispatch by default, so it can catch some programmers out and wreak havoc in their programs by simply forgetting the keyword `virtual`.

In Ada, the equivalent functionality is again provided by controlled types, by overriding the procedure Finalize:

```
with Ada.Finalization;
package P is
   type T is new Ada.Finalization.Controlled with private;
   function Make return T;  -- constructor
private
   type T is ... end record;
   overriding procedure Finalize (This: in out T);  -- destructor
end P;
```

Because Finalize is a primitive operation, it is automatically "virtual"; you cannot, in Ada, forget to make a destructor virtual.

Encapsulation: public, private and protected members

In C++, the unit of encapsulation is the class; in Ada, the unit of encapsulation is the package. This has consequences on how an Ada programmer places the various components of an object type.

```
class C {
public:
   int a;
   void public_proc();
protected:
   int b;
   int protected_func();
private:
   bool c;
   void private_proc();
};
```

A way to mimic this C++ class in Ada is to define a hierarchy of types, where the base type is the public part, which must be abstract so that no stand-alone objects of this base type can be defined. It looks like so:

```
private with Ada.Finalization;

package CPP is

  type Public_Part is abstract tagged record  -- no objects of this type
    A: Integer;
  end record;

  procedure Public_Proc (This: in out Public_Part);

  type Complete_Type is new Public_Part with private;

  -- procedure Public_Proc (This: in out Complete_Type);  -- inherited, implicitly defined
```

```
private  -- visible for children

  type Private_Part;  -- declaration stub
  type Private_Part_Pointer is access Private_Part;

  type Private_Component is new Ada.Finalization.Controlled with record
    P: Private_Part_Pointer;
  end record;

  overriding procedure Initialize (X: in out Private_Component);
  overriding procedure Adjust     (X: in out Private_Component);
  overriding procedure Finalize   (X: in out Private_Component);

  type Complete_Type is new Public_Part with record
    B: Integer;
    P: Private_Component;  -- must be controlled to avoid storage leaks
  end record;

  not overriding procedure Protected_Proc (This: Complete_Type);

end CPP;
```

The private part is defined as a stub only, its completion is hidden in the body. In order to make it a component of the complete type, we have to use a pointer since the size of the component is still unknown (the size of a pointer is known to the compiler). With pointers, unfortunately, we incur the danger of memory leaks, so we have to make the private component controlled.

For a little test, this is the body, where the subprogram bodies are provided with identifying prints:

```
with Ada.Unchecked_Deallocation;
with Ada.Text_IO;

package body CPP is

  procedure Public_Proc (This: in out Public_Part) is  -- primitive
  begin
    Ada.Text_IO.Put_Line ("Public_Proc" & Integer'Image (This.A));
  end Public_Proc;

  type Private_Part is record  -- complete declaration
    C: Boolean;
  end record;

  overriding procedure Initialize (X: in out Private_Component) is
  begin
    X.P := new Private_Part'(C => True);
    Ada.Text_IO.Put_Line ("Initialize " & Boolean'Image (X.P.C));
  end Initialize;
```

```
   overriding procedure Adjust (X: in out Private_Component) is
   begin
     Ada.Text_IO.Put_Line ("Adjust " & Boolean'Image (X.P.C));
     X.P := new Private_Part'(C => X.P.C);  -- deep copy
   end Adjust;

   overriding procedure Finalize (X: in out Private_Component) is
     procedure Free is new Ada.Unchecked_Deallocation (Private_Part, Private_Part_Pointer);
   begin
     Ada.Text_IO.Put_Line ("Finalize " & Boolean'Image (X.P.C));
     Free (X.P);
   end Finalize;

   procedure Private_Proc (This: in out Complete_Type) is  -- not primitive
   begin
     Ada.Text_IO.Put_Line ("Private_Proc" & Integer'Image (This.A) & Integer'Image (This.B) & ' ' & Boolean'Image (This.P.P.C));
   end Private_Proc;

   not overriding procedure Protected_Proc (This: Complete_Type) is  -- primitive
     X: Complete_Type := This;
   begin
     Ada.Text_IO.Put_Line ("Protected_Proc" & Integer'Image (This.A) & Integer'Image (This.B));
     Private_Proc (X);
   end Protected_Proc;

end CPP;
```

We see that, due to the construction, the private procedure is not a primitive operation.

Let's define a child class so that the protected operation can be reached:

```
package CPP.Child is

   procedure Do_It (X: Complete_Type);  -- not primitive

end CPP.Child;
```

A child can look inside the private part of the parent and thus can see the protected procedure:

```
with Ada.Text_IO;

package body CPP.Child is

   procedure Do_It (X: Complete_Type) is
   begin
     Ada.Text_IO.Put_Line ("Do_It" & Integer'Image (X.A) & Integer'Image (X.B));
     Protected_Proc (X);
   end Do_It;
```

```
end CPP.Child;
```

This is a simple test program, its output is shown below.

```
with CPP.Child;
use  CPP.Child, CPP;

procedure Test_CPP is

  X, Y: Complete_Type;

begin

  X.A := +1;
  Y.A := -1;

  Public_Proc (X);  Do_It (X);
  Public_Proc (Y);  Do_It (Y);

  X := Y;

  Public_Proc (X);  Do_It (X);

end Test_CPP;
```

This is the commented output of the test program:

```
Initialize TRUE                    Test_CPP: Initialize X
Initialize TRUE                                and Y
Public_Proc 1                      |  Public_Proc (X):  A=1
Do_It 1-1073746208                 |  Do_It (X):        B uninitialized
Adjust TRUE                        |  |  Protected_Proc (X): Adjust local copy X of This
Protected_Proc 1-1073746208        |  |  |
Private_Proc 1-1073746208 TRUE     |  |  |  Private_Proc on local copy of This
Finalize TRUE                      |  |  Protected_Proc (X): Finalize local copy X
Public_Proc-1                      |  ditto for Y
Do_It-1 65536                      |  |
Adjust TRUE                        |  |
Protected_Proc-1 65536             |  |
Private_Proc-1 65536 TRUE          |  |
Finalize TRUE                      |  |
Finalize TRUE                      |  Assignment: Finalize target X.P.C
Adjust TRUE                        |  |           Adjust: deep copy
Public_Proc-1                      |  again for X, i.e. copy of Y
Do_It-1 65536                      |  |
Adjust TRUE                        |  |
Protected_Proc-1 65536             |  |
Private_Proc-1 65536 TRUE          |  |
```

```
Finalize TRUE                       |  |
Finalize TRUE                       Finalize Y
Finalize TRUE                            and X
```

You see that a direct translation of the C++ behaviour into Ada is difficult, if feasible at all. Methinks, the primitive Ada subprograms corresponds more to virtual C++ methods (in the example, they are not). Each language has its own idiosyncrasies which have to be taken into account, so that attempts to directly translate code from one into the other may not be the best approach.

De-encapsulation: friends and stream input-output

In C++, a friend function or class can see all members of the class it is a friend of. Friends break encapsulation and are therefore to be discouraged. In Ada, since packages and not classes are the unit of encapsulation, a "friend" subprogram is simply one that is declared in the same package as the tagged type.

In C++, stream input and output are the particular case where friends are usually necessary:

```cpp
#include <iostream>
class C {
public:
   C();
   friend ostream& operator<<(ostream& output, C& arg);
private:
   int a, b;
   bool c;
};

#include <iostream>
int main() {
   C object;
   cout << object;
   return 0;
};
```

Ada does not need this construct because it defines stream input and output operations by default: The default implementation of the `Input`, `Output`, `Read` and `Write` attributes may be overridden (shown for `Write` as an example). The overriding must occur before the type is frozen, i.e. (in the case of this example) in the package specification.

```
private with Ada.Streams;  -- needed only in the private part
package P is
   type C is tagged private;
private
   type C is tagged record
```

```
      A, B : Integer;
      C : Boolean;
   end record;
   procedure My_Write (Stream : not null access Ada.Streams.Root_Stream_Type'Class;
                       Item   : in C);
   for C'Write use My_Write;  -- override the default attribute
end P;
```

By default, the `Write` attribute sends the components to the stream in the same order as given in the declaration, i.e. A, B then C, so we change the order.

```
package body P is
   procedure My_Write (Stream : not null access Ada.Streams.Root_Stream_Type'Class;
                       Item : in C) is
   begin
      -- The default implementation is to write A then B then C; here we change the order.
      Boolean'Write (Stream, Item.C);  -- call the
      Integer'Write (Stream, Item.B);  --   default attributes
      Integer'Write (Stream, Item.A);  --      for the components
   end My_Write;
end P;
```

Now `P.C'Write` calls the overridden version of the package.

```
with Ada.Text_IO.Text_Streams;
with P;
procedure Main is
   Object : P.C;
begin
   P.C'Write (Ada.Text_IO.Text_Streams.Stream (Ada.Text_IO.Standard_Output),
              Object);
end Main;
```

Note that the stream IO attributes are not primitive operations of the tagged type; this is also the case in C++ where the friend operators are not, in fact, member functions of the type.

Terminology

Ada	C++
Package	class (as a unit of encapsulation)
Tagged type	class (of objects) (as a type) (*not* pointer or reference, which are class-wide)
Primitive operation	virtual member function
Tag	pointer to the virtual table
Class (of types)	a tree of classes, rooted by a base class and including all the (recursively-)derived classes of that base class
Class-wide type	-
Class-wide operation	static member function
Access value to a specific tagged type	-
Access value to a class-wide type	pointer or reference to a class

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 (http://www.adaic.com/standards/95lrm/html/RM-3-8.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-3-8.html)]
- 3.9: Tagged Types and Type Extensions (http://www.adaic.com/standards/95lrm/html/RM-3-9.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-3-9.html)]
- 3.9.1: Type Extensions (http://www.adaic.com/standards/95lrm/html/RM-3-9-1.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-3-9-1.html)]
- 3.9.2: Dispatching Operations of Tagged Types (http://www.adaic.com/standards/95lrm/html/RM-3-9-2.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-3-9-2.html)]
- 3.9.3: Abstract Types and Subprograms (http://www.adaic.com/standards/95lrm/html/RM-3-9-3.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-3-9-3.html)]
- 3.10: Access Types (http://www.adaic.com/standards/95lrm/html/RM-3-10.html) [Annotated (http://www.adaic.com/standards/95aarm/AARM_HTML/AA-3-10.html)]

Ada 2005

- 3.8: Record Types (http://www.adaic.com/standards/05rm/html/RM-3-8.html) [Annotated (http://www.adaic.com/standards/05aarm/html/AA-3-8.html)]
- 3.9: Tagged Types and Type Extensions (http://www.adaic.com/standards/05rm/html/RM-3-9.html) [Annotated (http://www.adaic.com/standards/05aarm/html/AA-3-9.html)]
- 3.9.1: Type Extensions (http://www.adaic.com/standards/05rm/html/RM-3-9-1.html) [Annotated (http://www.adaic.com/standards/05aarm/html/AA-3-9-1.html)]
- 3.9.2: Dispatching Operations of Tagged Types (http://www.adaic.com/standards/05rm/html/RM-3-9-2.html) [Annotated (http://www.adaic.com/standards/05aarm/html/AA-3-9-2.html)]
- 3.9.3: Abstract Types and Subprograms (http://www.adaic.com/standards/05rm/html/RM-3-9-3.html) [Annotated (http://www.adaic.com/standards/05aarm/html/AA-3-9-3.html)]

- 3.9.4: Interface Types (http://www.adaic.com/standards/05rm/html/RM-3-9-4.html) [Annotated (http://www.adaic.com/standards/05aarm/html/AA-3-9-4.html)]
- 3.10: Access Types (http://www.adaic.com/standards/05rm/html/RM-3-10.html) [Annotated (http://www.adaic.com/standards/05aarm/html/AA-3-10.html)]

Ada Quality and Style Guide

- Chapter 9: Object-Oriented Features (http://www.adaic.org/docs/95style/html/sec_9/)

New in Ada 2005

This is an overview of the major features that are available in **Ada 2005**, the version of the Ada standard that was 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 Amendment (http://www.ada-auth.org/AI-XREF.HTML#Amend_Doc) 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 (http://libre.adacore.com/)
- GCC 4.1 (http://gcc.gnu.org/)
- GNAT Pro 6.0.2 (http://www.adacore.com/home/gnatpro/) (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 Ada 2005 is the default mode in GNAT GPL 2007 Edition.

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 [8] (http://gcc.gnu.org/onlinedocs/gnat_ugn_unw/Character-Set-Control.html). 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:

- AI95-00285-01 Support for 16-bit and 32-bit characters (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00285.TXT)
- AI95-00388-01 Add Greek pi to Ada.Numerics (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00388.TXT)

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:

- AI95-00251-01 Abstract Interfaces to provide multiple inheritance (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00251.TXT)
- AI95-00345-01 Protected and task interfaces (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00345.TXT)

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:

- AI95-00216-01 Unchecked unions -- variant records with no run-time discriminant (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00216.TXT)

- Annex B.3.3 Pragma Unchecked_Union (http://www.adaic.com/standards/05rm/html/RM-B-3-3.html) (Annotated (http://www.adaic.com/standards/05aarm/html/AA-B-3-3.html))

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:

- AI95-00217-06 Limited With Clauses (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-50217.TXT)
- AI95-00262-01 Access to private units in the private part (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00262.TXT)

Access types

Not null access

An access type definition can specify that the access type can never be null.

See Ada Programming/Types/access#Null exclusions.

See also: AI95-00231-01 Access-to-constant parameters and null-excluding access subtypes (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00231.TXT)

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:

- AI95-00230-01 Generalized use of anonymous access types (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00230.TXT)
- AI95-00385-01 Stand-alone objects of anonymous access types (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00385.TXT)
- AI95-00318-02 Limited and anonymous access return types (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-10318.TXT)

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 AI95-00302-03 Container library (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-20302.TXT) has now been transformed into A.18 Containers (http://www.adaic.com/standards/05rm/html/RM-A-18.html) (Annotated (http://www.adaic.com/standards/05aarm/html/AA-A-18.html)).

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 AI95-00302-03 AI Text (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-20302.TXT) 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 (http://charles.tigris.org).

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 (http://www.stepanovpapers.com/) has a few publications available.

Scan Filesystem Directories and Environment Variables

See also:

- AI95-00248-01 Directory Operations (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00248.TXT)

- AI95-00370-01 Environment variables (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00370.TXT)

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:

- AI95-00388-01 Add Greek pi (π) to Ada.Numerics (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00388.TXT)
- AI95-00296-01 Vector and matrix operations (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00296.TXT)

(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:

- AI95-00297-01 Timing events (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00297.TXT)
- AI95-00307-01 Execution-Time Clocks (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00307.TXT)
- AI95-00354-01 Group execution-time budgets (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00354.TXT)
- AI95-00266-02 Task termination procedure (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-10266.TXT)
- AI95-00386-01 Further functions returning Time_Span values (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00386.TXT)

Ravenscar profile

See also:

- AI95-00249-01 Ravenscar profile for high-integrity systems (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00249.TXT)
- AI95-00305-01 New pragma and additional restriction identifiers for real-time systems (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00305.TXT)
- AI95-00347-01 Title of Annex H (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00347.TXT)

- AI95-00265-01 Partition Elaboration Policy for High-Integrity Systems (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00265.TXT)

New scheduling policies

See also:

- AI95-00355-01 Priority Specific Dispatching including Round Robin (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00355.TXT)
- AI95-00357-01 Support for Deadlines and Earliest Deadline First Scheduling (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00357.TXT)
- AI95-00298-01 Non-Preemptive Dispatching (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00298.TXT)

Dynamic priorities for protected objects

See also: AI95-00327-01 Dynamic ceiling priorities (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00327.TXT)

Summary of what's new

New keywords

Added 3 keywords (72 total)

- `interface`
- `overriding`
- `synchronized`

New pragmas

Added 11 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

Added 7 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.Wide_Bounded_IO
  - Ada.Wide_Text_IO.Wide_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.Wide_Wide_Bounded_IO
  - Ada.Wide_Wide_Text_IO.Wide_Wide_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/Ada 83
- Ada Programming/Ada 95
- Ada Programming/Ada 2012
- 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:Book:Ada Programming/Ada 2005 feature

External links

Papers and presentations

- Ada 2005: Putting it all together (http://www.sigada.org/conf/sigada2004/SIGAda2004-CDROM/SIGAda2004-Proceedings/Ada2005Panel.pdf) (SIGAda 2004 presentation)
- GNAT and Ada 2005 (https://www.adacore.com/books/gnat-and-ada-2005) (SIGAda 2004 paper)
- An invitation to Ada 2005 (http://sigada.org/ada_letters/sept2003/Invitation_to_Ada_2005.pdf), and the presentation of this paper (http://www.cs.kuleuven.ac.be/~dirk/ada-belgium/events/04/040616-aec-ada2005.pdf) at Ada-Europe 2004

Rationale

- *Rationale for Ada 2005 (http://www.adaic.com/standards/05rat/html/Rat-TTL.html)* by John Barnes:

1. Introduction
2. Object Oriented Model
3. Access Types
4. Structure and Visibility
5. Tasking and Real-Time
6. Exceptions, Generics, Etc.
7. Predefined Library
8. Containers
9. Epilogue

References
Index

Available as a single document for printing (http://www.adaic.com/standards/05rat/Rationale05.pdf).

Language Requirements

- *Instructions to the Ada Rapporteur Group from SC22/WG9 for Preparation of the Amendment to ISO/IEC 8652* (http://www.open-std.org/jtc1/sc22/WG9/n412.pdf) (10 October 2002), and a presentation of this document (http://std.dkuug.dk/JTC1/sc22/wg9/n423.pdf) at SIGAda 2002

Ada Reference Manual

- **Ada Reference Manual**, ISO/IEC 8652:1995(E) with COR.1:2001 and AMD.1:2007 (http://www.adaic.com/standards/05rm/html/RM-TTL.html)
- **Annotated Ada Reference Manual**, ISO/IEC 8652:1995(E) with COR.1:2001 and AMD.1:2007 (http://www.adaic.com/standards/05aarm/html/AA-TTL.html) (colored diffs)
- List of Ada Amendment drafts (http://www.ada-auth.org/amendment.html)

Ada Issues

- Amendment 200Y (http://www.ada-auth.org/AI-XREF.HTML#Amend_Doc)
  - AI95-00387-01 Introduction to Amendment (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00387.TXT)
  - AI95-00284-02 New reserved words (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-10284.TXT)
  - AI95-00252-01 Object.Operation notation (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00252.TXT)
  - AI95-00218-03 Accidental overloading when overriding (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-20218.TXT)
  - AI95-00348-01 Null procedures (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00348.TXT)
  - AI95-00287-01 Limited aggregates allowed (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00287.TXT)
  - AI95-00326-01 Incomplete types (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00326.TXT)
  - AI95-00317-01 Partial parameter lists for formal packages (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00317.TXT)
  - AI95-00376-01 Interfaces.C works for C++ as well (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00376.TXT)
  - AI95-00368-01 Restrictions for obsolescent features (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00368.TXT)
  - AI95-00381-01 New Restrictions identifier No_Dependence (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00381.TXT)
  - AI95-00224-01 pragma Unsuppress (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00224.TXT)
  - AI95-00161-01 Default-initialized objects (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00161.TXT)
  - AI95-00361-01 Raise with message (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00361.TXT)

- AI95-00286-01 Assert pragma (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00286.TXT)
- AI95-00328-01 Preinstantiations of Complex_IO (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00328.TXT)
- AI95-00301-01 Operations on language-defined string types (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00301.TXT)
- AI95-00340-01 Mod attribute (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00340.TXT)
- AI95-00364-01 Fixed-point multiply/divide (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00364.TXT)
- AI95-00267-01 Fast float-to-integer conversions (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00267.TXT)
- AI95-00321-01 Definition of dispatching policies (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00321.TXT)
- AI95-00329-01 pragma No_Return -- procedures that never return (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00329.TXT)
- AI95-00362-01 Some predefined packages should be recategorized (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00362.TXT)
- AI95-00351-01 Time operations (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00351.TXT)
- AI95-00427-01 Default parameters and Calendar operations (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00427.TXT)
- AI95-00270-01 Stream item size control (http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00270.TXT)

Containers

Ada Programming/Containers

Interfacing

Ada Programming/Interfacing

Coding Standards

Ada Programming/Coding standards

Tips

Ada Programming/Tips

Common Errors

Ada Programming/Errors

Algorithms

Ada Programming/Algorithms

Chapter 1

Ada Programming/Algorithms/Chapter 1

Chapter 6

Ada Programming/Algorithms/Chapter 6

Knuth-Morris-Pratt pattern matcher

Ada Programming/Algorithms/Knuth-Morris-Pratt pattern matcher

Binary Search

Algorithm implementation/Search/Binary search

Function overloading

Ada Programming/Function overloading

Mathematical calculations

Ada Programming/Mathematical calculations

Statements

Ada Programming/Statements

Variables

Ada Programming/Variables

Lexical elements

Ada Programming/Lexical elements

Keywords

Ada Programming/Keywords

Delimiters

Ada Programming/Delimiters

Operators

Ada Programming/Operators

Attributes

Ada Programming/Attributes

Pragmas

Ada Programming/Pragmas

Libraries

Ada Programming/Libraries

Libraries: Standard

Ada Programming/Libraries/Standard

Libraries: Ada

Ada Programming/Libraries/Ada

Libraries: Interfaces

Ada Programming/Libraries/Interfaces

Libraries: System

Libraries: GNAT

Ada Programming/Libraries/GNAT

Libraries: Multi-Purpose

Ada Programming/Libraries/MultiPurpose

Libraries: Container

Ada Programming/Libraries/Container

Libraries: GUI

Ada Programming/Libraries/GUI

Libraries: Distributed Systems

Ada Programming/Libraries/Distributed

Libraries: Databases

Ada Programming/Libraries/Database

Libraries: Web

Ada Programming/Libraries/Web

Libraries: Input Output

Ada Programming/Libraries/IO

Platform Support

Ada Programming/Platform

Platform: Linux

Ada Programming/Platform/Linux

Platform: Windows

Ada Programming/Platform/Windows

Platform: Virtual Machines

Ada Programming/Platform/VM

Portals

Ada Programming/Portals

Tutorials

Ada Programming/Tutorials

Web 2.0

Ada Programming/Web 2.0
