Ada
Distilled

An Introduction to Ada Programming Features
for
Experienced Computer Programmers

by

Richard Riehle

AdaWorks Software Engineering
http:/www.adaworks.com

Version: Februrary 2002

Acknowledgments

There are always a lot of people involved in the creation of any book, even one as small and modest as this one. Those who have contributed to the best features of this book include my students at Naval Postgraduate School, Mr. Michael Berenato of Computer Sciences Corporation, Mr. Ed Colbert of Absolute Software, and many students from Lockheed-Martin Corporation, Computer Sciences Corporation, British Aerospace, various branches of the uniformed services, to name a few. I also owe a special thanks to Dr. Ben Brosgol, Dr. Robert Dewar, Mr. Mark Gerhardt, and Dr. Mantak Shing for what I have learned from them. Also thanks to the contributors to comp.lang.ada Usenet forum and the Team_Ada Listserve. Phil Thornley deserves extra credit for his detailed reading of the manuscript and many corrections.

Special thanks goes to Ed Colbert for his careful study of some of my program examples. He is one of those people who can spot a program error at fifty paces. Using this unique skill, Ed brought many errors, some big and some small, to my attention. Also thanks to more recent input from Phil Thornley and Adrian Hoe.

Any other errors are strictly mine. Any mistakes in wording, spelling, or facts are mine and mine alone.

I hope this book will be valuable to the intended audience. It is moderate in its intent: help the beginning Ada programmer get a good start with some useful examples of working code. More advanced books are listed in the bibliography. The serious student should also have one of those books at hand when starting in on a real project.

Richard Riehle

Audience for this Book

This book is aimed at experienced programmers who want to learn Ada at the programming level. It is not a "...for dummies" book, nor is it intended as a program design book. Instead, we highlight some key features of the Ada language, with coded examples, that are essential for getting started as an Ada programmer.

Ada is a rich and flexibile language used designing large-scale software systems. This book emphasizes syntax, control structures, subprogram rules, and how-to coding issues rather than design issues. There are some really fine books available that deal with design. Also, this is not a comprehensive treatment of the Ada language. The bibliography lists some books targeted toward comprehensive treatment of the language.

Think of this a quick-start book, one that enables you, the experienced programmer to get into the Ada language quickly and easily.

Happy Coding,

Richard Riehle

Table of Contents

1. What is Ada Distilled?

This little book is for the newcomer to Ada. The intended audience is experienced programmers rather than designers. Example programs are commented so an experienced programmer can experiment with Ada. The programmer who knows another language and wants annotated examples will find this helpful. This is not a comprehensive book on the entire Ada language. Many Ada features are ignored. In particular, we say very little about Ada.Finalization, Storage Pool Management, Representation Specifications, Dynamic Binding, Polymorphism, Concurrency, and other more advanced topics. Other books, listed in the bibliography, cover advanced topics. This book is an entry point to your study of Ada.

The text is organized around example programs with line by line comments. A comment might be an explanatory note and/or corresponding section of the Ada Language Reference Manual (ALRM) in the format of ALRM X.5.3/22. So you might see,

```
with Ada.Text_IO;                        -- 1  10.1.2, A.10 Context clause
procedure  Do_This is                    -- 2  6.3          Specification with "is"
begin                                    -- 3  6.3          Start algorithmic code
     Ada.Text_IO.Put_Line(“Hello Ada”);  -- 4  A.10.6       Executable source code
end Do_This;                             -- 5   6.3          End of procedure scope
```

where each line is numbered and the 10.1.2 and 6.3, etc. refer to ALRM Chapter 6.3 and ALRM Chapter 10.1.2, and A.10.6 refers to Annex A.10.6. There is occasional commentary by source code line number.

1.1 Ada Compilers and Tools

Ada 95 compilers are available for a wide range of platforms. A free compiler, GNAT, based on GNU technology, can be downloaded from the Web. A partial list of commercial sources for compilers includes Ada Core Technologies (GNAT), DDC-I, Rational, RR Software, Irvine Compiler Corporation, Green Hills, Aonix, and OC Systems.

Development tools are coming into existence at a fairly fast pace. At present, there are nearly a dozen different offerings for developing programs on Microsoft operating systems. There are also GUI development tools such as GtkAda for developing Ada software targeting platforms such as Microsoft operating systems, Linux, BSD, OS/2, Java Virtual Machine, and every variety of Unix.

1.2 Ada Education

The bibliography of this book lists some of the books and educational resources availble to the student of Ada. Some colleges and universities that offer Ada courses. In addition, companies such as AdaWorks Software Engineering where this author is employed, provide classes for corporations engaged in Ada software development. You can also find public classes in Ada for industry students. The bibliography of this book list publications and Internet sources where you can improve your knowledge of Ada.

1.3 Ada's Reputation

There is a lot of misinformation about Ada. One misconception is that it is a large, bloated language designed by committee. This is not true. Ada is designed around a few simple principles that provide the framework for the language design. Once you understand these principles, Ada will be as easy (if not easier) as many other languages. We highlight some of those principles in this book. One important principle is that the Ada compiler never assumes anything. You, the programmer, must always be precise.

2. Summary of Language

Ada is not an acronym. It is the name of the daughter of the English Poet, Lord Byron. She is credited with being the "first computer programmer" because of the prescience demonstrated in her early writings that described Charles Babbage's Analytical Engine. She was honored for this contribution by having a language named after her.

2.1 Goals and Philosophy

Every programming language is intended to satisfy some purpose, some set of goals. Sometimes the goals are defined in terms of a programming paradigm. For example, a goal might be to design an object-oriented programming language. Another goal might call for a language that conforms to some existing programming model with extensions to satisfy some new notions of programming techniques. Ada's goals are defined in terms of the final product of the software process, rather than to satisfy an academic notion of how programs should be designed and written. Ada's Goals are:

- **High reliability and dependability for safety-critical environments**
- **Maintainable over a long span by someone who has never seen the code before**
- **Emphasis on program readability instead of program writeability,**
- **Capability for efficient software development using reusable components**

In summary, Ada is designed to maximize the amount error checking a compiler can do as early in the development process as possible. Each syntactic construct is intended to help the compiler meet this goal. This means some Ada syntax may seem extraneous but has an important role in tipping-off the compiler about potential errors in your code. The default for every Ada construct is *safe*. Ada allows you to relax that default when necessary. Contrast Ada's default of *safe* with most of the C family of languages where the default is usually, *unsafe*.

Another important idea is *expressiveness* over *expressibility*. Nearly any idea can be expressed in any programming language. That is not good enough. Ada puts emphasis on expressiveness, not just expressibility. In Ada, we map the solution to the problem rather than the problem to the solution.

2.2 Elementary Syntax

The syntax of Ada is actually easy to learn and use. It is only when you get further in your study that you will discover its full power. Just as there is "no royal road to mathematics," there is no royal road to software engineering. Ada can help, but much of programming still requires diligent study and practice.

2.2.1 Identifiers

Identifiers in Ada are not case sensitive. The identifier Niacin, NIACIN, NiAcIn will be interpreted by the compiler as the same. Underbars are common in Ada source code identifiers; e.g. Down_The_Hatch. There is a worldwide shortage of curly braces. Consequently, Ada does not use { and }. Also, Ada does not use square braces such as [and] . Ada has sixty-nine reserved words. Reserved words will usually be shown in bold-face type in this book. (*See Appendix A for a complete list of reserved words*).

2.2.2 Statements, Scope Resolution, Visibility

Ada's unique idea of visibility often causes difficulties for new Ada programmers. Once you understand visibility, nearly everything else about Ada will be clear to you

An Ada statement is terminated with a semicolon. The entire scope of a statement is contained within the start of that statement and the corresponding semicolon. Compound statements are permitted. A compound statement has an explicit ***end*** of scope clause. A statement may be a subprogram call, a simple expression, or an assignment statement.

```
X := C * (A + B);                          --  1  Simple assignment statement
Move (X , Y);                              --  2 A procedure call statement
if A = B then                              --  3 Start a compound if statement
   J := Ada.Numerics.Pi * Diameter;        --  4 Compute the circumference of a circle
else                                       --  5 Part of compound if statement
   J := Ada.Numerics.Pi * Radius ** 2;     --  6 Compute area of a circle
end  if;                                   --  7 End of compound statement scope
if  (A and B) or  ((X and T)  and  (P or Q)) then --  8 Parentheses required in mixed and/or construct
   Compute(A);                             --  9 Call Compute subprogram
else                                       -- 10 Part of compound statement
   Compute(P);                             -- 11 Subprogram call statement
end  if;                                   -- 12 End of compound statement scope
```

Note on Line 8 that an Ada conditional statement cannot mix ***and*** and ***or*** unless the expression includes parentheses. This eliminates problems associated with such expressions. It also eliminates arguments about precedence of mixed expressions, and errors due to incorrect assumptions about precedence.

2.2.3 Methods (Operators and Operations)

Methods in Ada are subprograms (procedure/function) and include both operators and operations. Operators include the symbols: =, /=, <, >, <=, >=, &, +, -, /, *. Other operators are the reserved words, ***and***, ***or***, ***xor***, ***not***, ***abs***, ***rem***, ***mod***. A designer is permitted to overload operators. Operators for a named type may be made visible through the ***use type*** clause. They can also be made visible through local renaming of the operator. For detailed operator rules, see ALRM 4.5.

One operation, *assignment* uses the compound symbol: :=. The := operation is predefined for all non-limited types. Assignment cannot be directly overloaded. Assignment is never permitted for limited types. A type may be limited in one view and non-limited in another view.

Other operations may be defined by the Ada programmer. These other operations are usually defined within a package specification. Operations are usually implemented as subprograms (procedures or functions).

Another operation is the membership test, not considered an operation by the language. Membership test uses the reserved word **in**. The word **in** can be combined with the word **not** to produce a negative membership test, **not in**. Membership testing is permitted for every Ada type, including limited types.

2.3 Library and Compilation Units

A single library unit may be composed of more than one compilation unit. This is called separate compilation. Ada ensures that separately compiled units preserve their continuity in relationship to related units. That is, date and time checking, library name resolution, and date and time checking of compiled units ensures every unit is always in phase with every other related complation and library unit

2.3.1 Library Units

An Ada program is composed of *library units*. A library unit is a unit that can be referred to using a ***with*** clause. The technical name for the *with* clause is *context clause*. A *context clause* is a little like a *#include* compiler directive in other languages, but with important differences. A library unit, before being placed in scope through a *context clause*, must have been successfully compiled. Once compiled, it is placed in a [sometimes virtual] Ada compilation library. A *context clause* does not make any of the elements of a library unit visible. Instead, a *context clause* simply puts those elements in scope, making them potentially visible. Library units may be composed of more than one *compilation unit.*

A library unit may be a *package* or a *subprogram*. Subprograms are either *functions* or *procedures*.

1.	**package**	*A collection of resources with something in common, usually a data type.*
2.	**procedure**	*A simple executable series of declarations and associated algorithmic code.*
3.	**function**	*An executable entity which always returns a data type result.*
4.	**child unit**	*A package, procedure, or function that is a child of a package.*

An Ada library unit consists of a specification part and implementation part. The implementation is sometimes called a ***body***. For a subprogram the specification part could be coded as,

```
procedure Open (F : in out File);                -- Procedure specification; requires body.
function  Is_Open (F : File) return Boolean;     -- Function specification; requires body
```

C/C++ programmer note: An Ada subprogram specification is analogous to, but not identical to, a function prototype.

A package is a collection of services (public and private), usually related through some data type. Most Ada library units will be packages. A package specification includes type declarations, subprograms (procedures and functions), and exceptions. Also, a package usually consists of a specification part (public and private) and an implementation part. The implementation part of a package is called the ***package body***. Rarely, one will see a package specification that does not require a body.

Here is a typical specification for a package library unit. Note that it has two parts. The public part is visible to a client of the package. The private part is never visible to a client.

```
package Machinery_1_3 is                                  --  1 Package specification; requires body
   type Machine is private;                               --  2 Specifies the visible part of the data type;
   procedure Turn_On  (M : in out Machine);               --  3 procedure specification
   procedure Turn_Off (M : in out Machine);               --  4 procedure specification
   function  Is_On (M : in Machine) return Boolean;       --  5 function   specification
private                                                   --  6 private part hidden from a client of contract
   type Machine is record                                 --  7 full definition of the publicly declared type
      Turned_On : Boolean := False;                       --  8 component of the type; OOP attribute
   end record;                                            --  9 scope terminator for the component
end Machinery_1_3;                                        -- 10 scope terminator for the specification
```

Public part

Private part

where a client of the package has visibility only to the public part. Here is a possible package body,

```
package body Machinery_1_3 is                             --  1 Package body; implements specification declarations
   procedure Turn_On  (M : in out Machine) is             --  2 Repeat procedure specification; compiler checks this
   begin                                                  --  3 Starts algorithmic section of procedure
      M.Turned_ON := True;                                --  4 Simple assignment statement of boolean value
   end Turn_On;                                           --  5 Procedure scope terminator is required

   procedure Turn_Off (M : in out Machine) is             --  6 Must match profile in specification
   begin                                                  --  7 Algorithms between begin and end
      M.Turned_On := False;                               --  8 M.Turned called dot notation
   end Turn_Off;                                          --  9 Name is optional but end is required

   function  Is_On (M : in Machine) return Boolean is     -- 10 In mode is like a constant; it may
   begin                                                  -- 11                  not be on left side of assignment
      return M.Turned_On;                                 -- 12 return statement required of every function
   end Is_On;                                             -- 13 Scope terminator for function
end Machinery_1_3;                                        -- 14 End of all declarations for this package
```

Body

Most often, the specification and the body are compiled separately. A specification must compile without errors before its body can be compiled. The Ada library manager will issue a fatal compilation error if the body is out of phase with the specification. A programmer creating a client of the package, has visibility only to the public part of the specification. The specification is a *contract* with a client of the package. The contract must be sufficient for the client to access the promised services. Every declaration in the specification must conform, exactly, the code in the body. The Ada compiler detects conformance to ensure consistency over the lifetime of a library unit. A change to a specification requires recompilation of the body. A change to the body does not require recompilation of the specification.

```
with Machinery_1_3;                                       --  1 Context clause. Puts Machinery_1_3 in scope
procedure Test_Machinery_1_3 is                           --  2 Specifxication for the procedure
     Widget : Machinery_1_3.Machine;                      --  3 Local object of type Machine
begin                                                     --  4 Starts the algorithmic section of this procedure
     Machinery_1_3.Turn_On (M => Widget);                 --  5 Call the Turn_On using dot notation and named association
     Machinery_1_3.Turn_Off (M => Widget);                --  6 Call the Turn_On using dot notation and named association
end Test_Machinery_1_3;                                   --  7 Scope of subprogram terminates with the end clause
```

A client of the package, such as Test_Machinery_1_3, never has visibility to the private part or the body of the package. Its only access is to the public part. However, the entire package is in scope, including the body. The body is completely hidden from all views from outside the package even though it in scope.

2.3.2 Compilation Units

Library units can be composed of smaller units called *compilation units*. The library unit is the full entity referenced in a *context clause*. An Ada package is usually compiled as two compilation units: package specification and package body. Do not think of a package specification as a C++ .h file. The specification can be compiled separately. Also, the package body does not *with* the specification. A package body can be further subdivided into even smaller compilation units called *subunits*. Subunits, used wisely, can have substantial benefits to the maintenance cycle of existing Ada programs.

The specification of Machinery_1_3 in the previous section can be compiled by itself. Later, the package body can be compiled. The procedure Test_Machinery_1_3 may be compiled before the package body of Machinery_1_3. The test program cannot be linked until all separately compiled units are compiled.

The package body for Machinery_1_3 could have been coded for separate compilation as,

```
 package body Machinery_1_3 is                                  -- 1
   procedure Turn_On  (M : in out Machine) is separate;         -- 2
   procedure Turn_Off (M : in out Machine) is separate;         -- 3
   function  Is_On (M : in Machine)                             -- 4
                            return Boolean is separate          -- 5
end Machinery_1_3;                                              -- 6
```

A subprogram declared *is separate* places a subunit in the library. The subunit may have its own context clauses, its own local variables, and its own algorithmic code. Also, each subunit may be compiled independently once its parent has been successfully compiled. This means easier, faster maintenance and better unit testing. During development, each subunit can be assigned to a different programmer

Compilation units in most Ada programs will be the package specification and package body. Sometimes, as in lines 2, 3, 5, you may see a subprogram specification compiled with a semicolon instead of an ... *is* ... *end* implementation. This implies separate compilation of the body for that specification.

Ada does not require separate compilation, but some Ada compilers do. An implementation is free to impose this requirement. The standards for most Ada development shops also require separate compilation.

Ada has a model for parent-child library units. A package, such as package Machinery, may be the root of a tree of child library units. This also provides a unique opportunity for separate compilation.

Here is an example of parent-child library units.

```
package Messenger is                                -- 1 Package specification; requires body
   type Message is private;                         -- 2 Visible part of the data type; name only
   function Create (S : String) return Message;     -- 3 function specification
   procedure Send     (M   in Message);             -- 4 procedure specification
   procedure Receive (M : out Message);             -- 5 procedure specification
   function  Size (M : in Message) return Natural;  -- 6 function specification
private                                             -- 7 private part hidden from a client of contract
   type Message is record                           -- 8 full definition of the publicly declared type
      Text : String (1..120) := (others => ' ');    -- 9 string component of the type; OOP attribute
      Length :  Natural := 0;                       -- 10 how many of the 120 values are in use
   end record;                                      -- 11 scope terminator for the component
end Messenger;                                      -- 12 scope terminator for the specification
```

Public view of specification (lines 1–7)

Private view of specification (lines 7–12)

```
with Ada.Calendar;
package Messenger.Dated is                          -- 1 Package specification; requires body
   type Dated_Message is private;                   -- 2 Visible part of the data type; name only
   function  Create (M : in Message)                -- 3 function specification
                         return Dated_Message;      -- 4 function always specifies a return type
private                                             -- 5 private part hidden from a client of contract
   type Dated_Message is record                     -- 6 full definition of the publicly declared type
      Text : Message;                               -- 7 string component of the type; OOP attribute
      Date  : Ada.Calendar.Time;                    -- 8 how many of the 120 values are in use
   end record;                                      -- 9 scope terminator for the component
end Messenger.Dated;                                -- 10 scope terminator for the specification
```

At first, this might be mistaken for a form of inheritance. It allows us to extend the original package and add another component. The experienced OOP practitioner will see that it is not inheritance; there is no *is_a* relationship. Instead, the declarative region for Messenger has been extended to include the declarations of Messenger.Dated. Any client of Messenger.Dated has direct visibility to the public declarations of Messenger. The private part of Messenger.Dated and the body of Messenger.Dated has direct visibility to the private and public parts of Messenger.

Dated_Message is implemented in a has_a relationship. This means that Dated_Message contains a value of type Message. Dated_Message cannot be converted to an object of type Message. They are two distinct types, even though one contains an instance of the other. We treat the subject of parent-child relationships in greater detail later in this book.

2.4 Scope and Visibility

Some programmers find the concept of visibility more difficult than any other part of Ada. Once they really understand visibility, everything else in language makes sense.

Failure to understand the difference between *scope* and *visibility* causes more problems for new Ada programmers than any other single topic. It is an idea central to the design of all Ada software. There is an entire ALRM chapter devoted to it, Chapter 8. A ***with*** clause puts a library unit into scope; none the resources of that unit are directly *visible* to a client. This is different from a #include in the C family of languages. Ada has several techniques for making elements directly visible, after they are placed in scope. Separating *scope* from *visibility* is an important software engineering concept. It is seldom designed into other programming languages. You will see examples coded in this book that illustrate this language feature. NOTE: ISO Standard C++ *namespace* adopts a weakened form of Ada's scope and visibility model.

2.4.1 Scope

Every statement and construct has an enclosing scope. Usually, the scope is easy to see in the source code because it has an entry point (declare, subprogram identifier, composite type identifier, package identifier, etc.) and an explicit point of termination. Explicit terminations are consistently coded with an ***end*** statement. Anytime you see an ***end*** clause, you know that is the closing point for some scope. Scope can be nested. For example, a procedure may be declared inside another procedure. Not as easy to notice is when a ***with*** statement (context clause) brings some library unit into scope. The context clause places all the resources of that library unit in scope, but makes none of those resources visible.

> *A pure interpretation of the scope mechanism might better describe this in terms of a declarative region. However, since this book is intended as an introduction to the practical aspects of the language, we limit our discussion to the somewhat more general view of this mechanism. For a more rigorous description, please consult the Ada LRM, Chapter 8.*

2.4.2 Visibility

In Ada, an entity may be in scope but not have direct visibility. This concept is more developed in Ada than in most programming languages. Throughout Ada Distilled you will see examples of visibility such as:

• use clauses	*makes all public resources of a package directly visible*
• use type clauses	*makes public operators directly visible for designated type*
• entity dot notation	*entity in notation is directly visible; usually the best option*
• renaming , locally, of operations/operators	*usually best option for making operators directly visible*

During development, an Ada compiler error message may advise you that some entity or other is not visible at the point where it is declared or used. Most often this visibility problem will relate to operators. You can use one of the mechanisms from the above list to make that entity visible.

Visibility will be illustrated throughout the examples in this book. It will be easier to demonstrate in the code examples than to trudge through a tedious jungle of prose.

2.5 Declarations, Elaboration, Dependencies

Most Ada software systems are composed of many independent components, most in the form of packages. These packages are associated with each other through context (with) clauses.

```
with A;
with B;
with C;
package Q is
```

```
with T;
package body Q is
  ...
end Q;
```

```
with A;
with B;
with C;
package R  is
```

```
with E;
with F;
package body R is
  ...
end R;
```

```
with R;
package T  is
  ...
end T;
```

```
with A;
package body T  is
  ...
end T;
```

Notice that dependencies between library units can be deferred to the package body. This a unique feature of Ada, based on the integral nature of packages but taking advantage of their separate compilation capability. This gives us the best of both capabilities. We can minimize the design dependencies by declaring context clauses for the package body instead for the package specification. This eliminates the need to re-compile (or re-examine) the relationships each time we make a change somewhere in our design.

An Ada program includes declarations and executable statements. The specification of a package is a set of declarations. The body of that package may also contain declarations. The scope of the declarations can be thought of as a *declarative region*. In the declarative region, declarations are in scope but not necessarily visible. In fact, declarations within a package body are in the declarative region, but are never visible to a client or child library unit.

Every Ada unit has, potentially, a place for declarations. These declarations must be elaborated before the program can begin its algorithmic part. Elaboration takes place without any action from the programmer, but Ada does provide some pragmas (compiler directives) to give the programmer some control over the timing and order of elaboration. Usually, elaboration occurs at execution time. A programmer may specify compile-time elaboration through pragma Preelaborate or pragma Pure. If that compile-time elaboration is possible, it will occur according to the semantics of each pragma.

The library units named in a context (with) clause must be elaborated before they are actually in scope for a client. When there are many context clauses, each must be elaborated. In some circumstances, resources of one library unit are needed to complete an action involving another library unit.

2.5.1 Ada Comb

An Ada program unit may sometimes be viewed in terms of the "Ada Comb," an idea first presented to me years ago by Mr. Mark Gerhardt. The Ada Comb demonstrates how declarations and algorithms are related within an implementation; i.e, subprogram body, task body, declare block, package body, etc.

```
kind-of-unit unit-name                 -- 1 procedure, function, package body, declare block, etc.
   local declarations                  -- 2 Must be elaborated prior to begin statement
begin                                  -- 3 Elaboration is done.  Now start executing statements
   handled-sequence-of-statements      -- 4 Handled because of the exception handler entry
exception                              -- 5 Optional.  Not every comb needs this.
   sequence-of-statements              -- 6 This is the area for exception handler code
end unit-name;                         -- 7 Every comb requires a scope terminator
```

Be conscious of the Ada Comb when studying the subprograms and algorithmic structures in this book. Local declarations may be any legal Ada code, except control structures and algorithms. Because Ada is a block-structured language, the local declarations may be other subprogram declarations (including their body), instances of types, instances of generic units, tasks or task types, protected objects or protected types, use clauses, compiler directives (pragma), local type declarations, constants, and anything else that falls into the category of the items just listed.

The *handled-sequence-of-statements* includes statements that operate on declarations. This includes assignment, comparisons, transfers of control, algorithmic code. More generally, this includes the three fundamental structures of the structure theorem (Jacopini and Böhm): sequence, iteration, selection. One may also embed a declare block, with its own local declarations, within the handled-sequence-of-statements.

```
with Ada.Text_IO;                              --  1 Is elaborated before being used
with Machinery;                                --  2 Is elaborated before being used
procedure Ada_Comb_Example_1 is                --  3 Name of enclosing unit
   Data :  Machinery.Machine;                  --  4 Declarations  local to enclosing unit
begin                                          --  5
   declare                                     --  6 Can declare local variables in this block
      Data : Integer := 42;                    --  7 The name, Data, hides the global declarations
   begin                                       --  8 Integer Data now is visible; Outer Data is not
      Data := Data + 1;                        --  9 Handled sequence of statements
   exception                                   -- 10 Start exception handler part of unit
      when some-exception =>                   -- 11 Name the exception after reserved word, when
         -- sequence of statements             -- 12 Any legal sequence of statements here
   end;                                        -- 13 End of scope of declare block
end Ada_Comb_Example_1;                        -- 14 End of enclosing scope
```

The Ada comb may be found in most units that contain algorithmic code. This includes procedures, functions, package bodies, task bodies, and declare blocks. Any of these units may include some kind of identifier. In production code, it is helpful to include the label at the beginning of the comb as well as at the end of it. Here is a variation on the previous example

```
procedure Ada_Comb_Example_2 is                --  1 Name of procedure
   Data :  Float := 0.0;                       --  2 Floating point declaration in scope
begin                                          --  3
   Integer_Block:                              --  4 A label for the declare block
   declare                                     --  5 Can declare local variables in this block
      Data : Integer := 42;                    --  6 The name, Data, hides the global declarations
   begin                                       --  7 Integer Data now is visible; Float Data is not directly visible
      Data := Data + 1;                        --  8 Simple incrementing statement
   exception                                   --  9 Localized exception handling region
      when Constraint_Error => ...             -- 10 Statements to handle the exception
   end Integer_Block;                          -- 11 Named end of scope for declare block
Data := Data + 451.0;                          -- 12 Float data is once more visible
end Ada_Comb_Example_2;                        -- 13 End of scope of procedure
```

The second example has an exception handler localized to the declare block. There is an identifier (label) for this declare block. A block label is any user-defined name followed by a colon. The block repeats the identifier at the end of its scope. In the scope of the declare block, the floating point variable with the same name as the item in the declare block is automatically made invisible, even though it still in scope. It could be made visible with dot notation (Ada_Comb_Example_2.Data ...). In general, avoid identical names within the same scope. In large-scale systems with many library units, avoiding this is not always possible.

This section covers basic syntax of Ada in the form of short, annotated programs. The annotations sometimes have ALRM references such as 13.3 (Chapter 13, Section 3) or A.10 (Annex A, Section 10).

2.6 Variables and Constants

A variable is an entity that can change its value within your program. That is, you may assign new values to it after it is declared. A constant, once it has been declared with an assigned value, is not permitted to change that value during its lifetime in your program. Variables and constants may be declared in a certain

place in your program, called the *declarative part*. Any variable must be associated with some *type*. The basic syntax for a declaration is,

```
name_of_variable : name_of_type;                -- for a scalar or constrained composite type
name_of_variable : name_of_type(constraint) ;   -- for an unconstrained composite type
```

Declarations for predefined types (*see package Standard in the appendices of this book*)

```
Value    : Integer;                   -- from Annex A, package Standard
Degrees  : Float;                     -- from Annex A, package Standard
Sentinel : Character;                 -- from Annex A, package Standard
Result   : Boolean;                   -- from Annex A, package Standard
Text     : String(1..120);            -- Must constrain a string variable
```

We could also initialize a variable at the time it is declared,

```
Channel   : Integer := 42;                             -- "...life, the universe, and everything."
Pi        : Float := Ada.Numerics.Pi;                  -- from Annex A.5, ALRM
ESC       : Character := Ada.Characters.Latin_1.ESC;   -- from Annex A, ALRM
Is_On     : Boolean := True;                           -- from Annex A.1, ALRM
Text      : String(1..120) := (others => '*');         -- Every element of Text initialized to asterisk
```

2.7 Operations and Operators

Ada distinguishes between operations and operators. Operators are usually the infix methods used for arithmetic, comparison, and logical statements. Operators are often a visibility problem for a new Ada programmer.

2.7.1 Assignment Operation

Somewhere among his published aphorisms and deprecations, Edsger Dijkstra observes that too few programmers really understand the complexities of the assignment statement. I have not been able to excavate the exact quote from those of his publications immediately at hand. It is true, however, that assignment is more and more complicated as new programming languages are invented. Ada is no exception, and may actually have more complicated rules about assignment than some other languages.

The Ada assignment operation is: **:=** a compound symbol composed of a colon symbol and equal symbol. It is legal for every Ada type except those designated as limited types. It is illegal, in Ada, to directly overload, rename, or alias the assignment operation. In a statement such as,

A := B + C * (F / 3);

the expression on the right side of the assignment operation is evaluated and the result of that evaluation is placed in the location designated by the variable on the left side. All the variables on both sides must be of the same type. In an expression,

Reminder: the assignment operator is legal only on non-limited types. Also, both sides of the assignment operator must conform to each other. Composite types must have the same size and constraints.

Note: Ada does not allow direct overloading of the assignment operator. Sometimes it is useful to do that kind of overloading, and Ada does have a facility for designing in this feature safely but indirectly, by deriving from a controlled type.

```
X := Y;
```

X and Y must both be of the same type. If they are not of the same type, the programmer may, under strictly defined rules, convert Y to a type corresponding to the type of X. An example of this is,

```
type X_Type is ...                  -- Ellipses are not part of the Ada language; used for simplification here
type Y_Type is ...
X := X_Type(Y);                     -- When type conversion is legal between the types
```

Type conversion is not legal between all types. If both types are numeric, the conversion is probably legal. If one type is derived from another, it is legal. Otherwise, type conversion is probably not legal.

Assignment may be more complicated if the source and target objects in the assigment statement are composite types. It is especially complicated if those composite types include pointers (access values) that reference some other object. In this case, access value components may create very entertaining problems for the programmer. For this reason, composite types constructed from pointers should be ***limited types***. For limited types, one would define a *Deep Copy* procedure. Ada makes it illegal to directly overload the assignment operator. Study an example of a deep copy in the generic Queue_Manager later in this book.

Sometimes two types are so completely different that assignment must be performed using a special generic function, Ada.Unchecked_Conversion. Do not be too hasty to use this function. Often there is another option. Note the following example:

```
with Ada.Unchecked_Conversion;                              --  1  Chapter 13 or ALRM
procedure Unchecked_Example is                              --  2  Generally speaking, don't do this
   type Vector is array (1 .. 4) of Integer;                --  3  Array with four components
   for Vector'Size use 4 * Integer'Size;                    --  4  Define number of bits for the array
   type Data is record                                      --  5
      V1,  V2, V3, V4 : Integer;                            --  6  A record with four components
   end record;                                              --  7
   for Data'Size use 4 * Integer'Size;                      --  8  Same number of bits as the array
   function Convert is new Unchecked_Conversion             --  9
                  (Source => Vector, Target => Data);       -- 10  Convert a Vector to a Data
   The_Vector : Vector := (2, 4, 6, 8);                     -- 11
   The_Data   : Data   := (1, 3, 5, 7);                     -- 12
begin                                                       -- 13
   The_Data := Convert(The_Vector);                         -- 14  Assignment via unchecked conversion
end Unchecked_Example;                                      -- 15
```

Even though Line 14 probably works just fine in all cases, many Ada practitioners will prefer to do the assignments one at a time from the components of Vector to the components of Data. There will be more code, but selected component assignment is guaranteed to work under all circumstances. Unchecked conversion may be less certain unless you are careful what you are doing.

2.7.2 Other Operations

There are several reserved words that could be considered as operations. Most of these such as ***`abort`***, ***`delay`***, ***`accept`***, ***`select`***, and ***`terminate`*** are related to tasking. Others include raise (for exceptions), ***`goto`***, and ***`null`***. Some Ada practitioners might not agree with the notion that these are operations, however, in any other language they would be so considered.

There are other operations, for non-limited types, predefined in Chapter Four of the Ada Language Reference Manual. Again, these might not be thought of as operations, but they do have functionality that leads us to classify them as operations. These include array slicing, type conversion, type qualification, dynamic allocation of access objects, and attribute modification (Annex K of ALRM).

Because Ada is based in object technology, the designer is allowed to create and overload other operators. Those operators are declared as subprograms: function and procedure specifications. The subprogram specifications (operations) are declared in the public part of a package specification. They are implemented in the body of a package. For example, in a stack package, the operations are Push, Pop, Is_Full, Is_Empty, etc. For abstract data types, the operations are typically described as subprograms on the type.

2.7.3 Operators

Ada distinguishes between operators and operations. This distinction is useful for visibility management. The operators are all of the infix logical operators (=, /=, <, >, <=, >=, **`and`**, **`or`**, **`xor`**), and some post-fix operators (**`abs`**, **`not`**), and arithmetic operators (+, -, *, /, **`rem`**, **`mod`**). These operators may be overloaded.

Operators can be thought of as functions. For example, for a type, T, function signatures might be:

```
function "="  (Left, Right : T) return Boolean;   -- signature for equality operator
function ">=" (Left, Right : T) return Boolean;   -- signature for equality operator
function "+"  (Left, Right : T) return T;         -- signature for addition operator
```

This same signature applies to all operators. The name of the operator is named in double quotes as if it were a string. You may write your own operators for your own types. There is a special visibility clause that makes all the operators for a named type fully visible:

`use type` *typename*;

Good software engineering practice suggest that one makes selected operators visible using the renames clause instead of the the **`use type`** clause. For example, if type T is defined in package P,

```
function "+"  (Left, Right : P.T) return P.T renames P."+";
```

2.8 Elementary Sequential Programs

There is a more in-depth discussion of this topic in Chapter

Subprograms, in Ada are of two kinds: ***procedures*** and ***functions***. A subprogram *may* be a standalone library unit. Often it is declared in some other unit such as a package specification. The implementation part of the subprogram is called the "body." The body for Open might be coded as:

```
procedure  Open(F : in out File) is                      -- Note the reserved word, is
   -- optional local declarations                         -- Between is and begin, local declarations
begin                                                     -- Subprogram body requires a begin
   -- some sequence of statements                         -- Some statements or reserved word null;
end Open; -- Most standards require repeating the identifier here  -- End required; Identifier optional but usual
```

Sometimes we code the subprogram specification and body together. We will see many cases of this in the example subprograms that follow. Recall from an earlier discussion that Ada separates the notion of *scope* from that of *visibility*. Also, remember that more Ada programmers have more trouble with visibility rules than with any other aspect of the language. Once you understand visibility, you will understand Ada.

2.8.1 Subprogram Parameters

Subprograms may have formal parameters. Formal parameters must have a *name*, a *type*, and a *mode*. A mode tells the compiler how a parameter will be used in a subprogram. There is one other kind of entity that looks like a procedure but has slightly different semantics: a task *entry*. The parameter ***mode*** may be **in**, **out**, **in out**, or **access**. We can simplify understanding of mode with the following table,

Mode	Function	Procedure	Assigment Operator Position
`in`	Yes	Yes	Only right side of := (a constant in subprogram)
`out`	No	Yes	Right or Left side of := (but has no initial value)
`in out`	No	Yes	Right or Left side of := (has initial value)
`access`	Yes	Yes	Only right side of := (but might assign to component)

Although the previous table is something of a over-simplification, it will work well for you as a programmer. Just understand that *out mode* parameters are not called with an initial value, and *access* mode parameters are pointing to some other data. The data being accessed may be modified even though the access value itself may not. Examples of parameters and their modes within a subprogram,

2.8.2 Subprogram Specifications with Parameters

```
procedure Clear (The_List : in out List);                      -- The_List can be on either side of :=
function Is_Empty (The_List : in  List) return Boolean;        -- The_List can be on right side of :=
function Is_Full (The_List : List) return Boolean;             -- default in mode
procedure Get (The_List : in List; Data : out Item);           -- two modes;  two parameters
procedure Set_Col (To : in Positive_Count := 1);               -- default value for in mode
procedure Update (The_List : in out List; Data : in Item);     -- two modes; two parameters
function  Item_Count (The_List : access List) return Natural;  -- The_List can be on right side of :=
procedure Item_Count (The_List : access List;                  -- The_List can be on allowed on right of :=
                      Count    : out Count);                   -- unitialized; left or right of :=
function  M_Data (Azimuth, Elevation, Time : Float) return Float;  -- Three parameters, same type
```

A call to a formal parameter with an actual parameter should usually include named association. Consider function M_Data, above. Which is more readable and more likely to be accurate?

```
R := M_Data (42.8, 16.2, 32.8);
R := M_Data (Elevation => 16.2, Time => 32.8, Azimuth => 42.8);
```

This kind of problem happens often in languages where there are three parameters of the same type. For example, in a C or C++ function,

```
int mdata (int x, int y, int z) { ... }
```

there is no easy way to ensure the right actual values are being sent to the right formal arguments

3. Types and the Type Model

3.1 Strong Typing

This is the language feature for which Ada is best known. It is not the only strong point in Ada, but it is the best known. The following examples will demonstrate how it works. A type, in Ada consists of four parts,

1. **A name for the type**
2. **A set of operations for the type**
3. **A set of values for the type**
4. **A wall between objects of one type and those of another type**

No structural equivalence as found in C, C++, and Modula. Strict name equivalence model. No automatic promotion of types from one level to another. Better type safety under these rules.

The last feature, the *wall*, is the default of the Ada typing model. Ada does provide capabilities for getting around or over the wall, but the wall is always there. There are two general categories of type, elementary and composite. A composite type is a record or an array. Everything else, for our purposes in this book, is an elementary type. (**Note**: there are minor exceptions to this rule when you get into more advanced Ada). Some types are predefined in a package Standard (see this Appendix A of this book). From the object-oriented viewpoint, a type has *state*, operations to *modify* state and operations to *query* state.

3.2 Type Safety

A better way to view strong typing is to think in terms of ***type safety***. Every construct in Ada is type safe. For Ada, type safety is the default. For most languages, type safe is not the default. In still other languages, type safety is an illusion because they support structural equivalence or implicit type promotion. Ada does not support either of those concepts because they are not type-safe. An Ada designer declares data types, usually in a *package* specification, with the constrained set of values and operations appropriate to the problem being solved. This ensures a solid contract between the client of a type and the promise made by the *package* in which the type is defined.

3.3 Declaring and Defining Types

3.3.1 Categories of types

Ada types can be viewed in two broad categories: ***limited***, and ***non-limited***. A type with a limited view cannot be used with the := expression, ever. All other types can be used with := as long as that assignment is between compatible (or converted view of) types. Ada defines certain types as always limited. These include task types, protected types, and record types with access discriminants.

Types in Ada may be considered in terms of their ***view***. A type may be defined with a ***public view*** which can be seen by a client of the type, and a ***non-public view*** that is seen by the implementation of the type. We sometimes speak of the ***partial view*** of the type. A partial view is a public view with a corresponding non-public view. Partial views are usually defined as private or limited private. Also, the public view of a type may be limited where the implementation view of that same type may be non-limited.

Another important category is ***private*** type versus ***non-private*** type. A limited type may also be private. A type with a private view may also have a view that is not private. Any Ada data type may have a view that is private with a corresponding view that is not private. The predefined operations for a non-limited private type include: := operation, = operator, /= operator. Any other operations for a private type must be declared explicitly by the package specification in which the type is publicly declared.

3.3.2 A Package of Non-private Type Definitions

In addition to predefined types declare in package Standard, the designer may also define types. These may be constrained or unconstrained, limited or non limited. Here are some sample type declarations.

```
package Own_Types is
   type Color is (Red, Orange, Yellow, Green, Blue, Indigo, Violet);          -- 1An enumerated type;
       -- an ordered set of values; not a synonym for a set of integer values -- 2 A single line comment
   type Farenheit is digits 7 range -473.0..451.0;                             -- 3 Floating point type
   type Money is delta 0.01 digits 12;                                         -- 4 Financial data type for accounting
   type Quarndex is range -3_000..10_000;                                      -- 5 Integer type; note underbar notation
   type Vector is array(1..100) of Farenheit;                                  -- 6 Constrained array type
   type Color_Mix  is array(Color) of Boolean;                                 -- 7 Constrained by Color set
   type Inventory is record                                                    -- 8 A constrained record type
      Description : String(1..80) :=  (others => '   ');                       -- 9 Initialized string type record component
      Identifier  : Positive;                                                  -- 10 A positive type record component
   end record;                                                                 -- 11End of record scope required by Ada
   type Inventory_Pointer is access all Inventory;                             -- 12 Declaring a pointer type in Ada
   type QData is array(Positive range <>) of Quarndex;                         -- 13 Unconstrained array type
   type Account is tagged record                                               -- 14 See next example: 1.5.3.3
      ID          : String (1..20);                                            -- 15 Uninitialized string type component
      Amount      : Money := 0.0;                                              -- 16 See line 4 of this package
   end record;                                                                 -- 17 Required by language
   type Account_Ref is access all Account'Class;                               -- 19 Classwide pointer type for tagged type
end Own_Types;
```

3.3.3 A Private type Package

Public view of specification

Private view of

```
package Own_Private_Types is                                            -- 1
   type Inventory is limited private;                                   -- 2 Partial definition of limited private type
   type Inventory_Pointer is access all Inventory;                       -- 3 Declaring a pointer type in Ada
   procedure Create(Inv : in out Inventory);                            -- 4 Create an empty instance of Inventory
   -- More operations for type Inventory                                -- 5
   type Account is tagged private;                                      -- 6 Partial definition of a tagged type
   type Account_Ref is access all Account'Class;                        -- 7 Classwide pointer type for tagged type
   procedure Create(Inv : in out Inventory);                            -- 8 Creates an empty Inventory record
   function Create (D : String; ID : Positive) return Account_Ref;      -- 9 returns access to new Inventory record
   -- More operations for tagged  type,  Account                        -- 10
private                                                                 -- 11Begin private part of package
   type Inventory is record                                             -- 12 A constrained record type
      Description : String(1..80) :=  (others => '   ');                -- 13Initialized string type record component
      Identifier  : Positive;                                           -- 14 A positive type record component
   end record;                                                          -- 15 End of record scope required by Ada
                                                                        -- 16
   type Account is tagged record                                        -- 17 Extensible record tagged type
      ID        : String(1..12);                                        -- 18 Uninitialized string type component
      Amount    : Float := 0.0;                                         -- 19 A float type record component
   end record;                                                          -- 20 Required by language
end Own_Private_Types;                                                  -- 21
```

Note the signature of the Create procedure on Line 4. Since the inventory type is *limited private*, we would often want the mode of parameter list to be **in out**. However, it is legal to have mode of **out** only.

3.4 Deriving and Extending Types

A new type may be derived from an existing type. Using the definitions from the previous package,

```
type Repair_Parts_Inventory  is new Inventory;          -- no extension  of  parent record is possible here
```

where Repair_Parts inherits all the operations and data definitions included in its parent type. Also,

```
type Liability is new Account                           -- 1 extended from tagged parent, lines 6, 17-20, above
```

```
    with record                                -- 2 required ;phrase for this construct
       Credit_Value : Float;                   -- 3 extends with third component of the record
       Debit_Value  : Float;                   -- 4 fourth component of the record
    end record;                                -- 5 record now extended with four elements
```

in which Liability inherits all the operations and components of its parent type but also adds two more components. This means that Liability now has four components, not just two. This is called extension of the type (extensible inheritance). From the list of declared types, one could have a access (pointer) variable,

```
 Current_Account  : Account_Ref;               --  Points to Account or Liability objects
```

which can point to objects of any type derived from Account. That is, any type in Account'Class. This permits the construction of heterogeneous data structures.

3.5 Operations on Types

Ada distinguishes between operators and operations. Operators include =, /=, <, >, <=, >=, **abs**, **and**, **or**, **xor**, +, -, *, /, **rem**, and **mod**. Operators may be overloaded. Operations include assignment and any named operation. Operations, except for the assignment operation, may also be overloaded.

Legal syntax for operations on types is defined in 4.5 of the ALRM. In general the rules are pretty simple. A limited type has no language-defined operations, not even the := (assignment) operation. Every other type has :=, at minimum. Private type and record operators include = and /=. All other types have operators =, /=, >, < , >=, <=, ***and***, ***or***, and ***xor***. The numeric types have operators +, -, *, /, and ***abs***. Integer numerics have ***rem*** and ***mod***. A designer may create operations for any type as necessary. A membership test, ***in***/***not in***, is legal for every type, including limited types.

3.6 Where to Declare a Type

Note: membership test not officially an operation or operator. It cannot be overloaded. It is available for limited types.

Usually, a type will be declared in a package specification along with its exported operations. Therefore,

```
 package Machinery is                              --  1  Package specification; requires body
    type Machine is private;                       --  2  Specifies the visible part of the data type;
    procedure Turn_On  (M : in out Machine);       --  3  procedure specification
    procedure Turn_Off (M : in out Machine);       --   4  procedure specification
    function  Is_On (M : in Machine) return Boolean;   --   5  function specification
    function  ">" (L, R : Machine) return Boolean; --  6  Declare the ">" function for private type
 private                                           --  7   private part hidden from a client of contract
    type Machine is record                         --  8   full definition of the publicly declared type
       Turned_On : Boolean := False;               --  9   component of the type; OOP attribute
    end record;                                    -- 10  scope terminator for the component
 end Machinery;                                    -- 11 scope terminator for the specification
```

will imply that the public operations available to a client of Machinery, for the type Machine, are:

- pre-defined assignment and test for equality and inequality
- procedures Turn_On and Turn_Off
- functions Is_On and ">"
- no other operations on type Machine are available in package Machinery.

Note: subprograms (procedures and functions) are analogous to methods or member functions in other languages. Most of the time these are public, but sometimes it is useful to make them private.

The language defined operations for a private type, Machine, are only assignment (: =) , Equality (=), and Inequality (/=). All other operations and operators for Machine must be explicitly declared in the contract, i.e., the package specification. The package has overloaded the ">" operator, so a client of this package can do a *greater than* compare on two machine objects.

3.7 The Wall Between Types

Note: by a "wall" we mean that values of differing types may not be directly mixed in expressions. Type conversion can sometimes help you across the wall. Other times, more roundabout approaches are required. This is in keeping with Ada's charter to be as type safe as

The fourth property for a type, the wall, is i

```
package Some_Types is                               --  1 Declare specification name
   type Channel is range 2..136;                    --  2 A constrained integer
   type Signal is new Integer                       --  3 Derived from Standard.Integer
                           range 1..150             --  4 with a range constraint
   type Level is digits 7;                          --  5 A floating point type
   subtype Small_Signal is Signal                   --  6 No wall with objects of type Signal
                     range 2..14;                   --  7 but smaller range than Signal
   type Color is (Red, Yellow, Green, Blue);        --  8 Enumerated type with four values
   type Light is (Red, Yellow, Green);              --  9 Another enumerated type
   type Traffic is new Color                        -- 10 Derived from Color but with a
                       range Red..Green;            -- 11            smaller range of values.
end Some_Types;
```

Warning. Most Ada practitioners recommend against this kind of package. It works well for our teaching example, but is poor design practice. Generally, a package should be designed so each type is accompanied by an explicit set of exported operations rather than depending on those predefined.

3.7.1 Type Rule Examples

The following procedure uses the package, Some_Types. It illustrates how the typing rules work. Therefore, this procedure will not compile for reasons shown. A corrected example will follow .

```
with Some_Types;                                                  --  1 No corresponding use clause; in scope only
procedure Will_Not_Compile is                                     --  2 Correct.  Too many errors for this to compile
 Ch1,  Ch2, Ch3   : Some_Types.Channel     := 42;                 --  3 Notice the dot notation in declaration
 Sig1, Sig2          : Some_Types.Signal      := 27;              --  4 Dot notatation makes type Signal visible
 Level_1, Level_2 : Some_Types.Level        := 360.0;             --  5 Dot notation again.  No use clause so this is required
 Tiny  : Some_Types.Small_Signal := 4;                            --  6
 Color_1, Color_2 : Some_Types.Color       := Some_Types.Red;     --  7Dot notation required here
 Light_1, Light_2  : Some_Types.Light       := Some_Types.Red;    --  8
 Tr1, Tr2, Tr3       : Some_Types.Traffic    := Some_Types.Red;   --  9
begin                                                             -- 10
   Ch3 := Ch1 + ch2;                                              -- 11 Cannot compile;  + operator not directly visible
   Level_1 := Ch1;                                                -- 12 Incompatible data types
   Tiny := Sig1;                                                  -- 13This is OK  because of subtype
   Color_1 := Light_1;                                            -- 14 Incompatible types in expression
   Light_2 := Tr1;                                                -- 15 Incompatible types
   Light_3 := Some_Types.Light(Color_1);                          -- 16 Type conversion not permitted for these types
   Tr3 := Color_1;                                                -- 17 Incompatible types
   Tr1 := Some_Types.Traffic'Succ(Tr2);                           -- 18 This statement is OK
end Will_Not_Compile;                                             -- 19
```

The following example corrects some of the problems with the preceding one. Note the need for type conversion. Also, we include an example of unchecked conversion. Generally, unchecked conversion is a bad idea. The default in Ada is to prevent such conversions. However, Ada does allow one to relax the default so operations can be closer to what is permitted in C and C++, when necessary.

```
with Some_Types;                                                  --  1 Context clause from prior example
with Ada.Unchecked_Conversion;                                    --  2 Context clause for generic  Ada library function
use  Ada;                                                         --  3 Makes package Ada directly visible
procedure Test_Some_Types is                                      --  4 Name for unparameterized procedure
 Ch1,  Ch2, Ch3    : Some_Types.Channel     := 42;                --  5 Initialize declared variables
 Sig1, Sig2           : Some_Types.Signal      := 27;             --  6 Note dot notation in declared variables
 Level_1, Level_2  : Some_Types.Level       := 360.0;             --  7 Declared variables with dot notation
 Tiny                   : Some_Types.Small_Signal := 4;           --  8
 Color_1, Color_2  : Some_Types.Color       := Some_Types.Red;    --  9 Enumerated type declarations
```

```
   Light_1, Light_2   : Some_Types.Light       := Some_Types.Red;          -- 10
   Tr1, Tr2, Tr3      : Some_Types.Traffic     := Some_Types.Red;          -- 11
   use type Some_Types.Channel;                                            -- 12 Makes operators visible for this type
   function Convert is new Unchecked_Conversion                            -- 13 Enable asssignment between variables of
      (Source => Some_Types.Light, Target => Some_Types.Traffic);          -- 14 differing types without compile-time checking
begin                                                                      -- 15
   Ch3 := Ch1 + ch2;                                                       -- 16 use type makes + operator visible
   Level_1 := Some_Types.Level(Ch1);                                       -- 17 Type conversion legal between numeric types
   Tiny := Sig1;                                                           -- 18 This will compile because of subtype
   Tr3 := Some_Types.Traffic(Color_1);                                     -- 19 OK. Traffic is  derived from Color
   Tr1 := Some_Types.Traffic'Succ(Tr2);                                    -- 21 This statement is OK
   Tr2 := Convert(Light_1);                                                -- 22 Assign dissimilar data without checking
  Light_2  := Convert(TR3);          -- Illegal   Illegal   Illegal        -- 23 Convert is only one direction
end Test_Some_Types;                                                       -- 24
```

Notice that operations are not permitted between incompatible types even if they have a set of values with identical names and internal structure. In this regard, Ada is more strongly typed than most other languages, including the Modula family and the C/C++ family. Type conversion is legal, in Ada, when one type is derived from another such as types defined under the substitutability rules of object technology.

3.7.2 Subtype Declarations

There is a slight deviation in orthogonality in meaning of subtypes in the Ada Language Reference Manual. This discussion relates to the reserved word, subtype, not the compiler design model.

Ada has a reserved word, ***subtype***. This is not the same as a subclass in other languages. If a ***subtype*** of a type is declared, operations between itself and its parent are legal without the need for type conversion.

```
procedure Subtype_Examples is                                              --  1 Subprogram specification
   type Frequency is digits 12;                                            --  2 Floating point type definition
   subtype Full_Frequency is Frequency range 0.0 .. 100_000.0;             --  3 subtype definition
   subtype High_Frequency is Frequency range  20_000.0 .. 100_000.0;       --  4 subtype definition
   subtype Low_Frequency is Frequency range  0.0 .. 20_000.0;              --  5 sutype definition
   FF : Full_Frequency := 0.0;                                             --  6 Variable declaration
   HF : Full_Frequency := 50_000.0;                                        --  7 Variable declaration
   LF : Full_Frequency :=  15_000.0;                                       --  8 Variable declaration
begin                                                                      --  9
   FF  := HF;                                                              -- 10 OK; no possible constraint error
   FF  := LF;                                                              -- 11 OK; no possible constraint error
   LF  := FF;                                                              -- 12 Legal, but potential constraint error
   HF  := LF;                                                              -- 13 Legal, but potential constraint error
end Subtype_Examples is                                                    -- 14
```

3.8 Elementary Types

Elementary types are of two main categories, ***scalar*** and ***access***. An access type is a kind of pointer and is discussed in Chapter 5 of this book. Scalar types are ***discrete*** and ***real***. Discrete types are enumerated types and integer types. Technically, integer types are also enumerated types with the added functionality of arithmetic operators. Numeric discrete types are signed and unsigned integers.

Non-discrete, real numbers include floating point, ordinary fixed point, and decimal fixed point. The Ada programmer never uses pre-defined real types for safety-critical, production quality software.

All scalar types may be defined in terms of precision and acceptable range of values. The designer is even allowed to specify the internal representation (number of bits) for a scalar value.

```
   type Index is mod 2**16                            -- an unsigned number type
   for Index'Size use 16                              -- allot sixteen bits for this type
   type Int16 is range -2 ** 15.. 2**15 - 1;          -- a signed integer number type
   for Int16'Size use 16;                             -- allot sixteen bits for this type
   type Int32 is range -2 ** 31 .. 2**31 - 1          -- a signed integer numeric type
   for Int32'Size use 32;                             -- allot 32 bits for this type
```

3.9 Composite Types

Composite types contain objects/values of some other type. There are four general categories of composite types: *arrays*, *records*, *task types*, and *protected types*. An array has components of the same type. A record may have components of different types. Task types and protected types are discussed later.

3.9.1 Arrays

An array may have components of any type as long as they are all the same storage size. Ada has three main options for array definition: anonymous, type-based unconstrained, type-based constrained. Other combinations are possible, but not discussed in this book. Ada allows true multi-dimensional arrays, as well as arrays of arrays. Two common formats for a one dimensional array are:

```
type Array_Type is array(Index_Type range <>) of Component_Type;   -- One dimensional unconstrained array
type Array_Type is array(Range_Constraint) of Component_Type;       -- One dimensional constrained array
```

Ada also has something called anonymous arrays. An anonymous array is less flexible than a typed array and cannot be passed as a parameter to a subprogram. We will not use them much in this book.

3.9.1.1 Array Procedural Example

The following procedure demonstrates a constrained array and an unconstrained array, along with declarations and some procedural behavior. The constrained array is a boolean array. We show this array because of its special properties when used with logical or, and, and xor. The unconstrained array simply demonstrates that an unconstrained array must be constrained before it may be used.

```
with Ada.Text_IO;                                                      -- 1 Context clause
use  Ada;                                                              -- 2 Visibility clause
procedure Array_Definitions is                                         -- 3
   package BIO is new Text_IO.Enumeration_IO(Enum => Boolean);         -- 5 IO package for Boolean type
   type Boolean_Set is array(1..4) of Boolean;                         -- 6 Constrained boolean array
   pragma Pack(Boolean_Set);                                           -- 7 Forces array to four bits
   for Boolean_Set'Alignment use 2;                                    -- 7.1 Align storage on 2 bytes
   type Float_Vector is array(Natural range <>) of Float;              -- 8 Unconstrained array
         -- Note that the index is of type Natural and can be any range of values from 0 through Integer'Last
   B1 : Boolean_Set := (True, True, True, False);                      -- 9
   B2 : Boolean_Set := (False, False, True, False);                    -- 10
   B3 : Boolean_Set := (True, True, False, True);                      -- 11
   F1 : Float_Vector(0..9) ;                                           -- 12
   F2 : Float_Vector(1..10);                                           -- 13
   procedure Display (Data : Boolean_Set; Comment : String) is         -- 14
    begin                                                              -- 15
      Text_IO.Put(Comment);                                            -- 16
      for I in Data'Range loop  -- Cannot run off the end of an array  -- 17
            BIO.Put(Data(I));                                          -- 18
            Text_IO.Put(" ");                                          -- 19
      end loop;                                                        -- 20
      Text_IO.New_Line;                                                -- 21
   end Display;                                                        -- 22
begin                                                                  -- 23
  F1(2) := F2(4);                                                      -- 24 Simple component assignment
  F1(5..7) := F2(6..8);   -- This is sometimes called "sliding"        -- 25 Assign slices of different sizes
  Display (B1, "B1 is "); Display(B2, "B3 is "); Display(B3, "B3 is ");  -- 26
  Display (B2, "B2 is ");                                              -- 27
```

*Bitwise Logical operators **and**, **or**, and **xor** may be used on a boolean array.*

procedure Display factors out the responsibility for displaying the results of the boolean operations in the body of this example.

```
    B3 := B1 and B2;                                   -- 28 Logical and of B1 and B2
    Display(B3, "B1 and B2 = ");                       -- 29
    B3 := B1 or B2;                                    -- 30 Logical or of B1 and B2
    Display(B3, "B1 or B2 = ");                        -- 31
    B3 := B1 xor B2;                                   -- 32 Logical xor of B1 and B2
    Display(B3, "B1 xor B2 = ");                       -- 33
end Array_Definitions;                                 -- 34
```

Line 8, in the previous program illustrates an unconstrained array. When an array is declared as unconstrained, a constrained instance of it is required before it can be used in an algorithm. Here are some other examples of one dimensional, arrays, constrained and unconstrained:

```
type Float_Vector is array(Natural range <>) of Float;        -- One dimensional unconstrained array
type Float_Vector is array(-473..451) of Float;                -- One dimensional constrained array
type Day is (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday);
type Float_Vector is array(Day) of Integer;                    -- One dimensional constrained array
```

Note that an array index can be any discrete type and does not have to begin with zero. Also, type String, defined in package Standard is defined as an unconstrained array with a Positive index type. All the operations permitted on ordinary arrays are also permitted on Strings.

3.9.1.2 Multi-dimensional Arrays

Ada allows both multiple-dimension arrays such as those found in Fortran or arrays of arrays such as those in the C family of languages. There is no language defined limit of number of dimensions. For example,

```
type Float_Matrix is array(Integer range <>, Positive range <>) of Float;     -- Two dimensional array
type Bool_Matrix is array (Natural range <>,                                   -- First dimension of three
                           Positive range <>,                                  -- Second dimension of three
                           Color    range <>) of Boolean;                      -- Third dimension of three
type Mat_Vector is array (Positive range <>) of Float_Matrix(1..20, 5..15);    -- One dimension of two dimensions
```

3.9.1.3 Array Initialization

In Ada, arrays may be initialized using a concept called an ***aggregate***. The word aggregate is not a reserved word, but it is an important part of the language. An unconstrained array may include an aggregate at the time it is constrained. Any array may be re-initialized with a new aggregate in the algorithmic part of a module. The rule is that an aggregate must be complete. That is, every component must be included in the aggregate. Here are some examples, using the definitions already shown in this section (2.5.9.1).

For one dimensional array:

See unconstrained array, Float_Vector, defined in the previous section.

```
V1 : Float_Vector (1..6) := (others => 0.0);                       -- Instance initialized to all 0.0
V2 : Float_Vector (1..3) := (1 => 12.3, 3 => 6.2, 2 => 9.4);       -- Instance with initial values
V3 : Float_Vector (0..120) := (0 => 2.6, 120 => 7.5, others  => 9.4);  -- others must appear last
V4 : Float_Vector (12..80) := (12 => 16.3, 20 => 6.2, others  => 1.5); -- Instance with initial values
V5 : Float_Vector (-473..-1) := (others  => Float'First);          -- Negative index range
```

In the above instances, V1 has six elements and is initialized to all 0.0, V2 has three elements and is initialized using named association. *Named association* allows the programmer to associate a component value with a named index. V3 has 121 elements. It is initialized using named association with an ***others*** option. V4 has 68 elements, starting with an index of 12.

In Ada, an integer type index value may begin anywhere in the number range. It may even be a negative value, as in example V5. The value of V4'First is 12. The index bound of V4'Range is 12 through 80.

For a two dimensional array:

```
M1 : Float_Matrix(1..10, 1..10) :=  ( 1   => (1  => 0.0,  others  => 1.0),        --  1 Named association for each
                                      10 => (10  => 0.0, others  => 1.0),         --  2 dimension of the array and
                                      others => (others => 1.0));                 --  3 others specified last
```

If you wanted to write a loop that would use Text_IO to display all of the values for M1 on a console, it might look like the following code,

```
for I in M1'Range(1)                                  --  1 Range(1) specifies first dimension of array
    loop                                              --  2 outer loop; should have been named
      for J in M1'Range(2)                            --  3 Range(2) specifies second dimension of array
        loop                                          --  4 Always name nested loops in production code
          Text_IO.Put(Float'Image(M1(I, J)) & "  " ); --  5 Convert component to text and print it
        end loop;                                     --  6
       Text_IO.New_Line;                              --  7 Carriage return/Line feed on display
    end loop;                                         --  8
```

3.9.1.4 Array Catenation

Some prefer the word concatenation; same idea.

One of the more useful operations on arrays is catenation. Catenation is predefined in the language using the ampersand (&) symbol. As with most operators, you may overload the catenator operator. The rules for catenation are in ALRM 4.5.3/4. Taking the Float_Vector, defined above, we can have the following:

```
V10 : Float_Vector (1..10) := V1 & V2 & 42.9;                        -- Catenate 42.9, V1 and V2
```

Often it is useful to catenate a value of a different type after converting it to an appropriate representation. Let's say we have a variable,

```
Bango : Integer := 451;   -- bango is the Japanese word for number.
```

Suppose we want to display the value of Bango on the video. We could do the following:

```
Ada.Text_IO.Put_Line("Paper burns at " & Integer'Image(Bango) & " Farenheit ");
```

This prints a string to the screen. The ampersand catenates the result of the image attribute (as if it were a built-in function) which in turn is catenated to the constant string, Farenheit, (notice the leading space to make formatting more readable). Attributes help to make Ada programs more portable.

3.9.2 Records

Ada records come in several forms, many of which are ignored in this book. Some of the forms such as variant records, unconstrained records, and discriminated records, are not important to the novice. This book is not concerned with advanced of seldom used language features. However, we will include a few examples of constrained records, some records with a single discriminants , and some tagged records for the student's future study.

Consider the following Ada package specfication that declares some record types.

```
package Record_Declarations is                  --  1 This specification would require a pragma Elaborate_Body
   type Library_Book is                         --  2 Simple constrained record
      record                                    --  3 reserved word, record
         ISBN : String (1..12);                 --  4 String component
         Title   : String(1..30);               --  5 String component
         Author : String(1..40);                --  6 String component
         Purchase_Price : Float;                --  7 Floating point component
         Copies_Available : Natural;            --  8 Subtype natural from package Standard
      end record;                               --  9 Must identify end of scope of each record
                                                -- 10
   type Message_1 is                            -- 11 Simple record with an
      record                                    -- 12             unconstrained data type
         Text : Unbounded_String;               -- 13 See ALRM A.4.5
         Length :  Natural;                     -- 14 See package Standard
      end record;                               -- 15
                                                -- 16
   type Message_2 (Size : Positive) is          -- 17 Record with a discriminant
      record                                    -- 18 This must be constrained before
         Text :  String(1..Size);               -- 19 it may be used.  Note that the Size
         Length :  Natural;                     -- 20 has a corresponding entry in the record
      end record;                               -- 21 Dynamically allocated records might not
                                                -- 22 be as efficient as you would like.
   type Message_3 (Size : Positive := 1) is     -- 23 Record with a default discriminant
      record                                    -- 24 This may be constrained or may use
         Text :  String(1..Size);               -- 25 the default constraint.  There are more
         Length :  Natural;                     -- 26 rules for this, but we defer them to an
      end record;                               -- 27 advancd discussion of the language
                                                -- 28xxxxxx
   type Message_4 is tagged                     -- 29 A tagged type.  This may be extended
      .record                                   -- 30 with more components
         Text : Unbounded_String;               -- 31 Unbounded String(See Ada.Fixed.Unbounded).
         Length :  Natural;                     -- 32
      end record;                               -- 33
                                                -- 34
   type Message_5 is new Message_4 with         -- 35 Derived from a tagged type and one
      record                                    -- 36 additional component.  This record now x
         Stamp : Calendar.Time                  -- 37 has a total of three components, those
      end record;                               -- 38 it inherits and the one defined within it.
                                                -- 39
   type Message_6 is                            -- 40 Record containing another record
      record                                    -- 41
         Message_Data : Message_1;              -- 42 See line 11
         Library_Data   : Library_Book;         -- 43 See line 2
      end record;                               -- 44.
end Record_Declarations;                        -- 45 This package might require a pragma Elaborate_Body
```

Note that some Ada practitioners believe this kind of record is not a good idea. Since the Size might be variable at run-time, each compiler will have a unique way of addressing how to best implement the code

The package, Record_Declarations, has no subprograms. Therefore, the rules of the language might require a special pragma (compiler directive) to advise the compiler that there is a package body.

Note that, on line 35, the type Message_5 is derived from and extended from Message_4. This is a form of inheritance. We could have the following:

```
M4 : Message_4;
M5 : Message_5;
  ...
M4 :=  Message_4(M5);     -- provide a Message_4 view of the object of derived type, Message_5
```

or

```
M5 := (M4 with Library_Book); -- extends M5 with necessary components during assignment
```

4. Control Structures for Algorithms

Even in an object-oriented language, there comes the point where we must actually code the algorithmic implementation. Ada has a rich set of algorithmic constructs that are easy to code and easy to read.

4.1 Iteration Algorithms in Ada

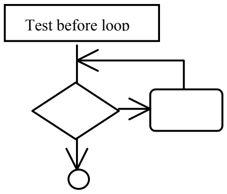
One of the three fundamental building blocks of every computer program is iteration. In nearly every serious program there is at least one loop. I realize some enthusiasts of recursion and/or functional programming (LISP, ML, CLOS, Haskell, etc.) may object to this statement.

4.1.1 For Loops

The famous proof in Italian by Jacopini and Bohm is important here since it is a foundation idea for program structure. From their proof, we understand the three fundamental control structures for imperative languages to be: sequence, iteration, and selection

A ***for loop*** is simple in Ada. Every ***loop*** must have an ***end loop***. The type of the index is derived from the type of the range variables. The scope of the index is the scope of the loop. The index is never visible outside the loop. Also, during each iteration of the loop, the index is a **constant** within the loop; that is, the index of a loop may not be altered via assignment. Iteration safety is fundamental to Ada.

```
with Ada.Integer_Text_IO;                          -- 1 Put Library Unit in Scope;                     A.10.8/21
procedure Sawatdee (Start, Stop : in Integer ) is  -- 2 "Good morning" in Thailand;                    6.2
begin                                              -- 3 Required to initiated sequence of statements
   for I in Start..Stop                            -- 4 I is a constant to the loop in each iteration;  5.5/9
      loop                                         -- 5 Reserved word loop is required;                 5.5
         Ada.Integer_Text_IO.Put(I);               -- 6 Note the use of "dot notation" to achieve visibility; A.10.8
      end loop;                                    -- 7 End loop is required for every loop;            5.5
end SaWatDee;   -- Ada is not case sensitive!      --  8 Note the label for the enclosing procedure;    6
```



An Ada enumerated type is an ordered set and may be used as the index of a loop. Also, the machine values for the enumerated type are not necessarily simple numbers as they are in C of C++. You will not need to do arithmetic on them. For an enumerated type, declared as:

```
type Week is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);  -- An enumerated type is an ordered set; (Sun < Mon)
```

consider the following loop.

```
with  Ada.Text_IO;                                 -- 1 Put Library Unit in Scope;  8.2, 10.1.2
procedure Dobroe_Uutra  is                         -- 2 "Good morning" in Russian
begin                                              -- 3 Required to initiated sequence of statements
    Loop_Name:                                     -- 4 This is a named loop;  good coding style; 5.5
    for Index in Week                              -- 5 Loop index may be any discrete type
        loop                                       -- 6 Reserved word loop is required;         5.5
           Ada.Text_IO.Put(Week'Image(Index));     -- 7 'Image converts Value to Text for  printing
        end loop Loop_Name;                        --  8 The name is required if the loop is named; 5.5
end Dobroe_Uutra;                                  --  9 Note the label for the enclosing procedure
```

Always label loops in production code. It helps with both maintenance and documentation

Next consider an anonymous array with a range from fifteen through sixty. We can traverse this with a simple loop statement and a 'Range attribute. There can be no indexing off the end of the array.

```
Set : array (15..60) of Integer;                   -- an anonymous array;  one of a kind;  no named type
```

consider the following loop with a loop label,

```
with Text_IO;                                      --  1  Put Library Unit in Scope
procedure Magandang_Umaga is                       --  2  "Good morning" in Tagalog (language of Phillipines)
begin                                              --  3  Required to initiated sequence of statements
   Outer:                                          --  4  This is a named loop;  good coding style
   for Index in  Set'Range                         --  5  Index'First = 15;  Index'Last = 60
```

```
        loop                                          --  6  Traverse the anonymous array
           Text_IO.Put(Integer'Image(Index));         --  7  'Image converts Integer to Text for printing
           Text_IO.Put_Line(Integer'Image(Set(Index))); --  8  Print the value in the array using 'Image
           Inner:                                     --  9  Give the inner loop a name
           for Day in Week loop                       -- 10  Note how we use type name for the range
             Text_IO.Put(Week'Image(Day));            -- 11  Convert the Day to Text for  printing
           end loop Inner;                            -- 12  The name of the loop is required
        end loop Outer;                               -- 13  The name is required if the loop is named
  end Magandang_Umaga;                                -- 14 Note the label for the enclosing procedure
```

Lines 7, 8, and 11 have code with the 'Image attribute. Check ALRM, Annex K/88 for details. Line 5 could have been coded as, **for** Index **in** Set'First .. Set'Last **loop** …

Sometimes you need to traverse a for loop in reverse. Line 5, above could have been coded as,

```
     for Index in  reverse Set'Range                 --  5  Cannot code:  for Index in 60..15 loop
```

A for loop might be used to traverse a two dimensional array. A nested loop will be required. Always label each loop when coding a nested loop. Here is the declaration of such an array.

```
type Matrix is array (Positive range <>, Natural range <>) of Integer;  -- an unconstrained Matrix

procedure Process (M : in out Matrix) is              -- 1 Specification for the procedure
begin                                                 -- 2 Simple begin
   Outer:                                             -- 3 Label for outer loop
   for  I in M'Range(1) loop                          -- 4 M'Range(1)  is first dimension of array
      Inner:                                          -- 5 Label for nested loop
      for  J  in M'Range(2) loop                      -- 6 M'Range(2) is second dimension
         -- do some actions on the matrix             -- 7 Algorithmic statements
      end loop Inner;                                 --  8 Inner end loop
   end loop Outer;                                    --  9 Outer end loop
end Process;                                          --  10 End of procedure scope
```

Always use loop labels when coding nested loop structures.

4.1.2 While Loops ALRM 5.5

A while loop is often the preferred type of loop in structured programming.

```
with Text_IO;                                                --  1 Put a library unit in scope
procedure Jo_Regelt is                                       --  2 "Good morning" in Hungarian
   The_File : Text_IO.File_Type;                             --  3 Declare internal file handle
   As_Input :  constant Text_IO.File_Mode := Text_IO.In_File; --  4 Is it input or output
   External_Name : String := "C:\Data\My.Txt";               --  5 Declare the external file name
   The_Data : String (1..80);                                --  6 A simple string variable;
   Line_Length : Natural;                                    --  7 For the input line parameter
begin                                                        --  8 Required to initiate a sequence of statements
   Text_IO.Open(The_File, As_Input, External_Name);          --  9 See Text_IO for  the types of the parameters
   Input_Routine:                                            --  10 You may name any kind of loop, and should!
   while not Text_IO.End_Of_File(The_File)                   --  11 Read The_File until finding the EOF mark
      loop                                                   --  12 Reserved word loop is required
         Text_IO.Get(The_File, The_Data, Line_Length);       --  13 Get a delimited string  from the file
         Text_IO.Put_Line(The_Data(1..Line_Length));         --  14 Echo the string with carriage / return line feed
      end loop Input_Routine;                                --  15 end loop name is required if the loop is named
end Jo_Regelt;                                               --  16 Note the label for the enclosing procedure
```

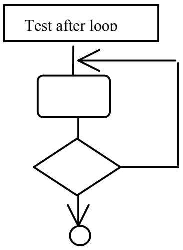
The following while loop uses the Get_Immediate feature of Ada.Text_IO, ALRM A.10.1/44.

```
with Ada.Text_IO;                                        --  1 Correct context clause
with Ada.Characters.Latin_1;                             --  2 Replaces Ada 83 package ASCII
procedure Hello_By_Input is                              --  3 Long procedure name
 ESC   : Character renames Ada.Characters.Latin_1.Esc;   --  4 A.3.3/5; Ada is not case sensitive
```

```
   Input  : Character := Ada.Characters.Latin_1.Space;        -- 5 Initial value for Variable
   Index : Natural := 0;                                       -- 6 package Standard,   A.1/13
   Hello : String(1..80) := (others => Input);                 -- 7 Input is intialized as space
begin                                                          -- 8 Normally comment this line
   Ada.Text_IO.Get_Immediate(Input);                           -- 9 ALRM A.101./44
   while Input /= ESC loop   -- /= is Ada "not equal" symbol   -- 10 Negative condition while loop
       Ada.Text_IO.Put(Input);  -- Echo input                  -- 11 Only Echo if it is not ESC
       Index := Index + 1;                                     -- 12 Need to maintain own index
       Hello(Index) := Input;                                  -- 13 Assign the input to the string
       Ada.Text_IO.Get_Immediate(Input);                       -- 14 No need to press enter key
  end loop;                                                    -- 15 Every loop needs an end loop
 Ada.Text_IO.New_Line;                                         -- 16 Carriage Return/ Line Feed
 Ada.Text_IO.Put_Line(Hello);                                  -- 17 Put the string and advance one line
end Hello_By_Input;                                            -- 18 Must be same name as procedure
```

Notice that this loop could be coded to avoid the ***while*** condition and simply do an ***exit***. This would eliminate the initial Get_Immediate on Line 9 but would require an ***if*** statement to effect the exit. Sometimes we want to **exit** a loop before we reach the pre-defined conditions. This can be used for a loop with no conditions or a loop in which some associated value goes abnormal. It can also be used to emulate the Pascal ***repeat*** ... ***until*** construct. There are several forms of the exit: ***exit when***, ***if*** *condition* ***then exit***, and the simple unconditional ***exit***. For each form, the careful programmer will include the name of the loop.

4.1.3 Exit Loop ALRM 5.7



```
with Text_IO;                                              -- 1 Put a library unit in scope
procedure Salaam_Ahlay_Kham  is                            -- 2 Parameterless declaration
      The_File : Text_IO.File_Type;                        -- 3 Declare internal file handle
      As_Input : Text_IO.File_Mode := Text_IO.In_File;     -- 4 Is it input or output
      External_Name : String := "C:\Data\My.Txt";          -- 5 Declare the external file name
      The_Data : String(1..80) := (others => ' ');         -- 6 Constrained, initialized string
      Line_Length : Natural;                               -- 7 For the input line parameter
begin                                                      -- 8 Required to initiated sequence of statements
      Text_IO.Open(The_File, As_Input, External_Name);     -- 9 See Text_IO for  the types of the parameters
      Controlled_Input:                                    -- 10 You may name any kind of loop, and should
      loop                                                 -- 11 Unconditional loop statement
            Text_IO.Get(The_File, The_Data, Line_Length);  -- 12 Get a delimited string  from the file
            exit Controlled_Input                          -- 13 Note the use of the label name
                  when  The_Data(1..2) = "##";             -- 14 A conditional exit;  should always be labled
            Text_IO.Put_Line(The_Data(1..Line_Length));    -- 15 Print the string with carriage return/line feed
      end loop Controlled_Input;                           -- 16 The name is required if the loop is named
end Salaam_Ahlay_Kham;                                     -- 17 Note the label for the enclosing procedure
```

Pay attention to line 10 in this example. A loop label makes this kind of loop easier to maintain. Many Ada practitioners suggest you never use an exit without a label. For consistency checking, the compiler will require the name of the loop at the end loop statement if there is a label. Here is some alternative syntax for lines 13 through 14 of the loop in P5, above,

```
if The_Data(1..2) = "##" then                   -- 13 An if statement to control the exit
      exit Controlled_Input;                     -- 14 Exit with a  label name
else                                             --
   ...                                           --
end if;                                          --
```

The syntax and rules of the if statement are discussed in the next section.

4.2 Selection Statements

Selection comes in two flavors. There is the alternation form, usually represented as an ***if*** ...***end if***, and the multiway selection, often coded as a ***case*** ... ***end case***. As is true of every elementary structure, there is an entry point and a well-defined end of scope. The end of scope is coded with an "end *kind-of-selection*".

4.2.1 If Statements ALRM 5.3

The basic if statement in Ada is not very complicated. There is a rule that every if must have an "end if." Also, unlike a language such as Pascal, an if condition may be compound. There is a reserved word, elsif, which permits a kind of multi-way condition selection. The following function is somewhat contrived, but it does illustrate the idea of the if along with the elsif. The most important thing to observe about elsif is that it might drop through all conditions if none are true. Therefore, you will almost always want a final else, even though it is not required by the language. If you fall through all possibilities in a function you may never reach a return statement which will cause the RTE to raise a Program_Error (ALRM, A.1/46) as an exception.

```
function Select (A,B,C : Float) return Float  is          --  1 Parameterized  function
     Result : Float : = 0.0;                               --  2 Local Variable for return statement.
begin                                                      --  3 Required to initiated sequence of statements
     if A > B then                                         --  4 Simple logical comparison
          Result := A  ** 2;                               --  5 Exponentiation of A; 4.5.6/7
     elsif  A < B then                                     --  6 Note the spelling;4.5.2/9
          Result := B ** 2;                                --  7 4.5.6/7
     elsif A <= C then                                     --  8 4.5.2/9
          Result := C * B;                                 --  9 4.5.5
     else                                                  -- 10 Optional else; but always include it
          Result := C * A;                                 -- 11 4.5.5
     end if;                                               -- 12 Try to have only one return statement.
     return Result;                                        -- 13 If no return is found, Program_Error is raised
end Select;                                                             -- 14 Always label a subprogram end
statement
```

The ***if*** statement is legal for nearly every Ada data type. Some types are designated as limited. Limited objects have no predefined equality or relational testing but do permit membership ***if*** tests.
record types and private types have predefined ***if*** tests for equality and membership. The creator of a limited type may define an equality or relational operator. For a private type or record the designer may overload equality or define a relational operator. Sometimes it is better to create an entirely new operation such as Is_Equal or Is_Greater For example, using the data type, Inventory, defined earlier.

```
function "=" (L, R : Inventory) return Boolean;         --  Specify an equality operator; operator overloading
function Is_Equal (L, R : Inventory) return Boolean;  --  Specify an equality operation;  Could be more readable
function ">" (L, R : Inventory) return Boolean;         --  Specify an greater-than operator
```

An implementation of "=" might look like this

```
function "=" (L, R : Inventory) return Boolean is        --  1 Redefines an equal operator
begin                                                    --  2 The usual begin statement
   return L.ID = R.ID;                                   --  3 Compare only the ID part.
end "=";                                                 --  4 Required scope terminator
```

An implementation of ">" might look like this

```
function ">" (L, R : Inventory) return Boolean is        --  1 Redefines ">" operator
begin                                                    --  2 The usual begin statement
   return L.ID > R.ID;                                   --  3 Compare only the ID part.
end "=";                                                 --  4 Required scope terminator
```

There is also a form of the ***if*** statement called short-circuit form. This takes two syntactic formats: ***`and then`*** and ***`or else`***. With the ***`and then`*** format , the programmer can explicitly indicate that if the comparison of the first operand fails, don't check the second operand. In the ***`or else`*** format, if the expression in the first operand is not TRUE, evaluate the second operand. If it is TRUE, then don't bother to evaluate the second operand.

4.2.2 Membership Testing 4.5.2/2

Tip: This is one of those powerful Ada syntactic constructs that can make code more readable and easier to

Sometimes you want a simple membership test. The ***in*** and ***not in*** options permit testing a range or even the membership of a value within a type or type range. A membership test is permitted for any data type. It often makes your ***if*** statements more readable.

```
function Continue(Data : Item) return Boolean  is          -- 1 Parameterized  function
    Result : Boolean  :=  False;                            -- 2 Initialized return variable.
begin    -- Continue                                        -- 3 Comment the begin statement
    if  Data in 1..20 then                                  -- 4 Simple membership test for a range
        Result := True;                                     -- 5 Set the result
    end if;                                                 -- 6 Always need an end if
    return Result;                                          -- 7 At least one return statement; required
end Continue;                                               -- 8 Always label the end statement
```

or for a data type derived from another type

```
type Bounded_Integer is new Integer range -473..451;   -- Derived type; derived from Standard Integer
```

```
procedure Demand                                        -- 1 Procedure Identifier
  (Data : in out Bounded_Integer'Base) is               --  2 Parameter list for Base type
    Local : Bounded_Integer'Base := 0;                  --  3 Initialized variable.
begin    -- Demand                                      --  4 Comment the begin statement
    Data := Data + Local;                               --  5 Comment the begin statement
    if Data in Bounded_Integer then                     --  6 Simple membership test for a range
        null;                                           --  7 Some Action
    end if;                                             --  8 Always need an end if
end Demand;                                             --  9  label the end statement
```

4.2.3 Case Statements ALRM 5.4

Ada ***case*** statements are easy and consistent. Unlike pathological case constructs in the C family of languages, Ada never requires a “break” statement. A case statement only applies to a discrete type such as an integer or enumerated type. Also, when coding a case statement, all possible cases must be covered. The following case statement illustrates several of these ideas. Consider an enumerated type, Color defined as:

```
type Color  is (White, Red, Orange, Yellow,  Chartreuse, Green,        -- The values are the names of the
                    Blue, Indigo, Violet, Black,  Brown),              --  colors. No need for numerics
```

The following function evaluates many of the alternatives.

```
function Evaluate (C : Color) return  Integer is          --  1 Simple function declaration
    Result : Integer := 0; -- I like to initialize everything  --  2 Local variable
begin  -- Evaluate                                         --  3 Comment the begin statement
    case C is                                              --  4 Start a case statement
        when Red => Result := 1;                           --  5 The => is an association symbol
        when Blue =>Result := 2;                           --  6 Am I blue? Set result to 2
        when Black .. Brown => Result := 3;                --  7 For black through brown ...
        when Orange | Indigo => Result := 4;               --  8 For either orange or indigo
        when  others => Result := 5;                       --  9 others required for unspecified cases.
    end case;                                              -- 10 Must use others if any cases are not specified
```

```
    return Result;                           -- 11 Compiler will look for a return statement
end Evaluate;                                -- 12 As usual, label the end statement
```

Sometimes, when a case statement result requires a long sequence of statements, consider using a ***begin**..* ***end*** block sequences (*see above discussion on blocks*). Always label a ***begin**..**end*** block.

```
function  Decide (C : Color) return  Integer is    --  1 Simple function declaration
    Result : Integer := 0;                         --  2 Local variable
begin  -- Decide                                   --  3 Comment the begin statement
    case C is                                      --  4 Start a case statement
        when Red =>                                --  5 One of the enumerated values
            begin                                  --  6  An unlabeled begin ... end sequence; see 4.4
               -- sequence-of-statements           --  7 Any sequence of Ada statements
            end;                                   --  8 Unlabeled end statement
        when Blue =>                               --  9 One of the enumerated values
            Label_1:                               -- 10 Better style; use a block label
            begin                                  -- 11 Alternative:  consider calling nested subprogram
               -- sequence-of-statements           -- 12 A labeled begin requires label name at end
            end Label_1;                           -- 13 The label is required for the end statement
        when  others =>                            -- 14  Ada requires others if some choices are unmentioned
            Label_2:                               -- 15 Yes. Still using the label; label an embedded begin block
            begin                                  -- 16
              -- handled-sequence-of-statements    -- 17 We expect a local exception handler.
            exception                              -- 18 This is a good use of begin...end blocks
              -- sequence-of-statements            -- 19 The exception handling statements
            end Label_2;                           -- 20 The compiler will look for this
    end case;                                      -- 21 Scope terminator is required
    return Result;                                 -- 22 Compiler will look for a return statement
end Decide;                                        -- 23 As usual, label the end statement
```

On line 14, the ***when others*** is required when some possible choices are not explicitly stated. An Ada compiler checks for the label at the end of a labeled begin..end block. If there is a ***when others*** and there are no other choices, the compiler issues an error message. Lastly, a choice may be stated only once. If you repeat the same choice, the Ada compiler will pummel you about the head and shoulders soundly.

4.3 Blocks

As shown in the preceding example, Ada allows the programmer to label in-line blocks of code. Sometimes these are labled loops. Other times they are simply short algorithmic fragments. A block may even include localized declarations. This kind of block is called a "declare block." Some Ada programming managers think in-line declare blocks are a reflection of poor program planning. In spite of that, they appear often in production code. In fact, a declare block is the only way to declare a local variable for a code fragment.

4.3.1 Begin ... End Blocks ALRM 5.6

This is a useful feature of Ada for trapping exceptions and sometimes for debugging. Good coding style suggests that they be labeled. Some Ada practitioners suggest using a labeled begin end with a case statement as noted in Section 3.3.3 of this book.

```
with Ada.Text_IO,                                   --  1 Note the comma instead of semicolon
Ada.Integer_Text_IO;                                --  2 Predefined package for Integer I/O
function Get return Integer  is                     --  3 Parameterless function
    package IIO renames Ada.Integer_Text_IO;        --  4 Make the name shorter via renames clause
    package TIO renames Ada.Text_IO;                --  5 Make the name shorter
    Data : Integer := -0;                           --  6 In scope for all of  P8
```

```
    Try_Limit : constant := 3;  -- universal integer constant   --  7 A constant cannot be changed
    Try_Count : Natural := 0                                     --  8 Natural cannot be less than zero
begin                                                            --  9 Required to initiated sequence of statements
    Input_Loop:                                                  -- 10 Optional label for the loop
    loop                                                         -- 11 Required reserved word
        Try_Block:                                               -- 12 Always name a begin..end block
        begin                                                    -- 13 Start begin ... end block
            Try_Count := Try_Count + 1;                          -- 14 Increment a variable by one
            IIO.Get(Data)                                        -- 15 Convert external text to internal number
            exit Input_Loop;                                     -- 16 unconditional loop exit
        exception                                                -- 17 Placed between begin ... end sequence
            when TIO.Data_Error =>                               -- 18 Exception handling
                if Try_Count > Try_Limit then                    -- 19 Decide whether to exit the loop
                  Text_IO.Put_Line(“Too many tries);             -- 20 Because the Try_Count is too high
                  exit Input_Loop;                               -- 21 exit the loop
                end if;                                          -- 22 Every if requires an end if
        end Try_Block;                                           -- 23 The label is required if block is labeled
    end loop Input_Loop;                                         -- 24 Loop is labeled so label is required
    return  Data;                                                -- 25  One return statement for this function
end Get;                                                         -- 26 Always label a subprogram end statement
```

4.3.2 Declare Blocks ALRM 5.6

A ***declare*** block is an in-line block of code which includes some local declarations. The scope of the declarations ends with the ***end*** statement of the block. If any local name is the same as some other name in the enclosing scope, the local name is the only one directly visible.

```
with Text_IO;                                          --  1  Put a library unit in scope
procedure Tip_A is                                     --  2  Parameterless declaration
    Rare_E : Float := 2.72; -- natural number, e       --  3  A rare E; see ALRM A.5
    Data : Integer := 42;                              --  4  In scope for entire procedure
begin                                                  --  5  Required to initiate sequence of statements
    Text_IO.Put(Integer’Image(Data));                  --  6  What will print? Integer is converted to a string
    declare                                            --  7  begin a new scope (declarative region)
        Data : Float := 3.14; -- a short slice of pi   --  8  Hide visibility of Integer, Data; see ALRM A.5
    begin                                              --  9  [optionally Handled ] sequence of statements
        Text_IO.Put(Float’Image(Data));                -- 10 X‘Image is allowed for Floating Point
    end;                                               -- 11 A scope terminator is required
    Ada.Text_IO.Put(Float'Image(Rare_E));              -- 12 A long way to tip a rare e.
end Tip_A;                                             -- 13 Always include a unit name
```

You may want to access the Data from an outer scope within a declare block. Names in an outer scope, with names in conflict with those within a declare block, can be made visible with “dot notation.” It is sometimes observed that declare blocks can be used for *ad hoc* routines that someone forgot to design into the software. For this reason, some Ada practitioners recommend frugality when using them. Also, because declare blocks can be so easily sprinkled through the code, it is essential that production declare blocks are always labeled. The following declare block illustrates several of these points.

```
with Text_IO;                                       --  1 Put a library unit in scope
with Ada.Integer_Text_IO, Ada.Float_Text_IO;        --  2 Predefined numeric IO packages
with Ada.Numerics;                                  --  3 ALRM, Annex A.5
procedure P7 is                                     --  4 Parameterless declaration
    package IIO renames Ada.Integer_Text_IO;        --  5 Make the name shorter via a renames clause
    X : Integer := 42;                              --  6 In scope for entire procedure
begin                                               --  7 Required to initiate sequence of statements
    IIO.Put(X);                                     --  8 What will print?
    Local_Block:                                    --  9 Always name a declare block
    declare                                         -- 10 begin a new scope (declarative region)
        use Ada.Integer_Text_IO;                    -- 11 controversial localization of use clause
        X : Float := Ada.Numerics.Pi;               -- 12 Hide visibility of global Integer, P7.X
    begin                                           -- 13 [optionally Handled ] sequence of statements
        Put(X);                                     -- 14 Put is visible because of  “use clause”
```

```
      IIO.Put(P7.X);                          -- 15 Dot qualifier  makes Integer X visible
   end Local_Block;                           -- 16 Labeled end  name required for labeled block
end P7;                                       -- 17 Always label a subprogram end statement
```

Tip: Consider promoting a declare block to a local (nested) parameterless procedure in the declarations of the enclosing unit.   This is more maintainable.   It can be made more efficient with an inline pragma.

5. Access Types (Pointers)

We don't really have pointers in Ada. The use of the word pointers is simply to acknowledge a corresponding capability via access types. The important thing is that the default for access types is *safe*, unlike pointers in the C family of languages

5.1 Overview of Access Types

The British computing pioneer, Maurice Wilkes, is credited with inventing *indirection*. Indirection is a generalized notion of a pointer. According to Dr. Wilkes, "There is no problem in computer programming that cannot be solved by not adding yet one more level of indirection." Pointers, in many languages have been problematic. The C family of languages encourages one to do arithmetic on pointers, thereby creating some really tricky errors. Ada pointers, called access types, do not have default capability for pointer arithmetic. Java, to its credit, adopted some of the Ada philosophy on pointers. Whenever we use the term pointer in Ada, we really mean *access* type or access object. When we refer to an access type, we are referring to a pointer with a default a set of safe rules and no arithmetic operators.

There are three forms of access type.

Access Type Form	Terminology
• Access to a value in a storage pool	*storage pool access type*
• Access to a declared value	*general access type*
• Access to a supbprogram (procedure or function)	*access to subprogram type*

Every access type is type specific to some designated type.

Storage pool access types will require some kind of storage pool management since objects are dynamically allocated to an area of memory, possibly the "Heap." Ada does not require automatic garbage collection but some compilers may provide it. Otherwise, use the package System.Storage_Pools defined in ALRM Chapter 13.

```
type Reference is access Integer;            -- Can only point to predefined type Integer; storage pool  access type
type Float_Reference is access all Float;    -- Can only point to predefined type Float; general access type
type Container is limited private;           -- Defines a data type with limited format; ordinary limited type
type Container_Pointer is access all Container; -- Can only point to objects of type Container; access to a limited type
type Method is access procedure … ;          -- Points to a procedure with corresponding parameter profile
type Method is access function … ;           -- Points to function with corresponding parameter profile and return type
```

5.2 Storage Pool Access Type

A storage pool access type requires an associated set of storage locations for its allocation. This might be a simple heap operation, or the serious Ada programmer can override the operations in System.Storage_Pool to enable some form of automatic garbage collection within a bounded storage space.

```
with Ada.Integer_Text_IO;  use Ada;            --  1 Library package for Integer IO
procedure Access_Type_1 is                      --  2
   type Integer_Pointer is access Integer;     --  3 Storage pool access type
   Number : Integer := 42;                      --  4 Declared value
   Location : Integer_Pointer;                  --  5 Storage pool access value
begin                                           --  6
   Location := new Integer;                     --  7 The word new is an allocator
   Location.all := Number;                      --  8 all permits reference to the data being referenced
   Integer_Text_IO.Put(Location);               --  9 Illegal.  Location is not an Integer type
   Integer_Text_IO.Put(Location.all);           -- 10 Legal.  Location.all is data of Integer type
end Access_Type_1;                              -- 11
```

Line 3 declares a type that points [only] to objects of type Integer. It cannot point to any other type. There is no pointer type in Ada that allows one to point to different types (except for classwide types). Line 4 declares an object of the pointer type. It has no value. The default initial value is **null**. An Ada pointer can never point to some undefined location in memory. Line 7 uses the reserved word ***new***. In this context, ***new*** is an *allocator*. An allocator reserves memory, at run time, for an object of some data type. On Line 7, the address of that memory is assigned to the variable named Location. The pointer named Location is not an Integer. Instead, it points to a storage location that contains an integer.

Ada, by default, prohibits arithmetic on a pointer. The following statement is not allowed in Ada.

```
Location := Location + 1;   -- illegal.  No pointer arithmetic allowed
```

If one really needs to do pointer arithmetic, it is possible through a special packages from Chapter 13 of the ALRM, package System.Address_To_Access_Conversions and package System.Storage_Elements. In practice, pointer arithmetic is unnecessary.

Line 8 refers to Location.all. This how one refers to the data in the memory where Location points. Notice that Line 9 will be rejected by the compiler, but Line 10 would compile OK.

5.3 General Access Type

A general access type provides additional capabilities to the storage pool access type. It permits storage allocation like storage pool access types. It also allows access to declared objects when those objects are labeled ***aliased***. Returning the example above,

```
with Ada.Integer_Text_IO;  use Ada;              --  1 Library package for Integer IO
procedure Access_Type_2 is                       --  2
  type Integer_Pointer is access all Integer;    --  3 General access type; requires all
  N1 : aliased Integer := 42;                    --  4 Aliased declared value
  N2 : Integer := 360;                           --  5 Non-aliased declared value
  Location : Integer_Pointer;                    --  6 General access type value
begin                                            --  7
  Location := N1'Access;                         --  8 Point to value declared on Line 4
  Integer_Text_IO.Put(Location);                 --  9 Illegal.  Location is not an Integer type
  Integer_Text_IO.Put(Location.all);             -- 10 Legal.  Location.all is data of Integer type
  Location := N2'Access;                         -- 11 Illegal. N2 was not aliased
end Access_Type_2;                               -- 12
```

The first difference in this example is on Line 3. Integer_Pointer is a *general access type* because the declaration includes the word, **all**. The next difference is Line 4. N1 is an ***aliased*** declared value. A general access type may only reference aliased values. The reserved word, ***aliased***, is required under most circumstances. Tagged type parameters for subprograms are automatically aliased. Line 8 is a direct assignment to an aliased value. This is legal. Contrast this with Line 11, which is not legal. Do you see that Line 11 is not legal because N2, on line 5, is not aliased?

5.3.1 Preventing General Access Type Errors

There is a potential danger with direct assignment to pointers. This danger is present all the time in the C family of languages. What happens when a data item goes out of scope and still has some other variable pointing to it? Ada has compiler rules to prevent this. The following example illustrates this.

```
with Ada.Integer_Text_IO;  use Ada;              --  1 Library package for Integer IO
procedure Access_Type_3 is                       --  2
  type Integer_Pointer is access all Integer;    --  3 General access type; requires all
  Location : Integer_Pointer;                    --  4 General access type value
begin                                            --  5
  declare                                        --  6 A declare block with local scope
    N1 : aliased Integer := 42;                  --  7 Declare an aliased value locally
  begin                                          --  8
    Location := N1'Access;                       --  9 Point to value declared on Line 4
  end;                                           -- 10 End of declare block scope
end Access_Type_3;                               -- 11 Compilation failed!  Sorry about that. ☺
```

The Ada compiler will reject this program. The rule is that the general access type declaration must be at the same level (same scope) as its corresponding variables. If you look at this example carefully, you will see that, when the declare block leaves its scope, Location would still be pointing to a value that has disappeared. Instead of using 'Access on line 9, the programmer could have coded 'Unchecked_Access, thereby bypassing the compile-time checks. Wisdom would dictate thinking very carefully before resorting to the use of any "unchecked" feature of the language. The word "unchecked" means the compiler does not check the validity or legality of your code. It is almost always an unsafe programming practice.

While the accessibility rules (See 5.3.2) might seem a drawback, they are easily managed in practice. Often it is enough to simply declare a local general access type and use it in a call to appropriate subprograms. The following example shows how this could happen.

```
procedure Access_Type_4 is                               -- 1
   function Spritz (I : access Integer) return Integer is -- 2
   begin                                                 -- 3
      return I.all + 1;                                  -- 4
   end Spritz;                                           -- 5
begin                                                    -- 6
   declare                                               -- 7
      type Integer_Pointer is access all Integer;        -- 8
      Location : Integer_Pointer;                        -- 9
      N1 : aliased Integer := 42;                        -- 10
      N2 : Integer := 0;                                 -- 11
   begin                                                 -- 12
      Location := N1'Access;                             -- 13 Assign location of N1 to Location
      N2 := Spritz(Location);                            -- 14 Call function with access variable parameter
   end;                                                  -- 15
end Access_Type_4;                                       -- 16
```

Not good coding style. Avoid these kinds of side-effect statements. This is the one and only place where C++ can be more reliable than Ada because of the way C++ controls constants.

All uses of the general access type are localized and the lifetime of each entity is appropriate to the others. There will be no potential dangling references when the declare block leaves its scope.

On line 14, a local access variable is used to call a function that has an access parameter. The access parameter is anonymous. We may not assign a location to it. However, it would be legal to code.

```
I.all := I.all + 1;  -- N1 would also be incremented by 1
return I.all;
```

But this is a very naughty thing to do. Shame on you if you do it!

This code would change the actual value of what Location is pointing to. Avoid doing this sort of thing. If you were to print the value for both N1 and N2, you would see the number 43. Some practitioners consider this a side-effect. Side-effects are rare in Ada and usually considered bad programming style.

5.3.2 The Accessibility Rules

ALRM Section 3.10.2, paragraphs 3 through 22, describe the accessibility rules. The purpose of the rules is to prevent dangling references. That is, when a variable is no longer in scope, there should be no access value trying to reference it. This is checked by the compiler. Under some rare circumstances, it might not be checked until run-time.

The rules can be summarized in terms of the lifetime of the access type itself. An object referenced by the 'Access attribute may not exist longer that the the access type to which it applies. Also, if an object is referenced with the 'Access attribute, it must be able to exist as long as the access type. The following three examples illustrate the point.

```
procedure Accessibility_Problem_1 is          -- 1
  type Integer_Reference is access all Integer; -- 2 General access type in scope
  Reference : Integer_Reference;              -- 3 Access value in immediate scope
  Data : aliased Integer;                     -- 4 Data at the same accessibility level
begin                                         -- 5
  Reference := Data'Access;                   -- 6 OK because types and declarations
```

This example will work just fine. No data will be left dangling when the scope is exited. Lifetime of all entities is the same.

```
    end Accessibility_Problem_1;                               -- 7 are at the same accessibility level

    procedure Accessibility_Problem_2 is                       -- 1
      type Integer_Reference is access all Integer;            -- 2 General access type
      Reference : Integer_Reference;                           -- 3 Access value
    begin                                                      -- 4
      declare                                                  -- 5
        Data : aliased Integer;                                -- 6 An aliased integer value
      begin                                                    -- 7
        Reference := Data'Access;                              -- 8 Will not compile; at wrong level of
      end;                                                     -- 9 accessibility for corresponding types.
    end Accessibility_Problem_2;                               -- 10
```

This will not compile. When the program exits the declare block, an outer pointer named Reference would still be pointing to data that no longer existed. This is not simply a dangling reference. It is a reference to data that is no longer valid. The Ada compiler will not let you do this.

```
    procedure Accessibility_Problem_3 is                       -- 1
      type Integer_Reference is access all Integer;            -- 2
    begin                                                      -- 3
      declare                                                  -- 4
        Reference : Integer_Reference;                         -- 5
        Data : aliased Integer;                                -- 6
      begin                                                    -- 7
        Reference := Data'Access;                              -- 8
      end;                                                     -- 9
end Accessibility_Problem_3;                                   -- 10
```

This will not compile. You might think that putting the actual pointer in the same local scope as the data being reference would work. The rule is that access value named Reference must exist at least as long as the

5.4 Access to Subprogram Types

One of the problems with the Ada 83/87 standard for Ada was the unavailability of some kind of pointer capability for subprograms. The current Ada standard does permit this. The rules for formation of such an access type are rather simple. The rules for visibility and accessibility of access to subprogram types are often difficult to manage in one's design.

5.4.1 Declaring an Access to Subprogram Type

- The type must have a parameter list corresponding to the subprogram being accessed
- The return type of a function access type must match that of the function being accessed
- Variables of the type may access any subprogram with a conforming profile

Examples:

```
type Action is access procedure(Data : in out Integer);
type Channel is access procedure(M : in out Message; L : out Natural);

type Condition_Stub is access function (Expression : Boolean) return Boolean;
type Compute is access function (L, R : Float) return Float;
```

The signature (parameter profile) of each subprogram access type must exactly match any subprogram being accessed.

5.4.2 Using an access to Subprogram Type

5.4.2.1 A Procedure Example

The following example demonstrates how to create an array of procedures. This is often useful when you have multiple procedures with the same profile but different behaviors. In this example we have kept the behavior simple to avoid confusion. The astute reader will immediately see the possibilities.

```
with Ada.Integer_Text_IO;                                      --  1 ALRM Annex A
with Ada.Text_IO;                                              --  2 ALRM Annex A
use  Ada;                                                      --  3 Makes Ada directly visible
```

```
procedure Alternative_Actions is                                   -- 4 Name of enclosing procedure
   type Action is access procedure (Data : in out Integer);        -- 5 Access to subprogram definition
    procedure Process (D : in out Integer) is                      -- 6 Procedure with correct profile
   begin                                                           -- 7
      D := D + D;                                                  -- 8 Details;  procedure behavior
    end Process;                                                   -- 9 end of scope of procedure
   type Process_Set is array(1..10) of Action;                     -- 10 Array type  of access types
   Index :  Positive;                                              -- 11 Used for array index later
   Value :  Integer := 0;                                          -- 12 Used for actual parameter
   The_Process : Process_Set := (others => Process'Access);        -- 13 access object array with aggregate
begin                                                              -- 14
   loop                                                            -- 15
      Text_IO.Put("Enter Index(1..10): ");                         -- 16
      Integer_Text_IO.Get(Index);                                  -- 17
      exit when Index not in 1..10;                                -- 18 membership test for exit
      Text_IO.New_Line;                                            -- 19
      Text_IO.Put("Enter Integer Value: ");                        -- 20
      Integer_Text_IO.Get(Value);                                  -- 21
      The_Process(Index)(Data => Value);                           -- 22 Named association clarifies
      Text_IO.New_Line;                                            -- 23
      Text_IO.Put("The result for Index " & Positive'Image(Index)  -- 24
                   & "is"  & Integer'Image(Value));                -- 25
  end loop;                                                        -- 26
end Alternative_Actions;                                           -- 27
```

5.4.2.2 A function Example

The following function example has behavior similar to the previous example. It has been altered a little bit to illustrate some additional capabilities.

```
with Ada.Text_IO; use Ada;                                                      -- 1
   procedure Function_Access_Type is                                            -- 2
   type Real is digits 12;                                                      -- 3 Define a floating point type
   package FIO is new Text_IO.Float_IO(Num => Real);                            -- 4 Instantiate IO package
   function Method (D : in Real) return Real is                                 -- 5 function w/correct profile
    begin                                                                       -- 6
     return D + D;                                                              -- 7
   end Method;                                                                  -- 8
   type Compute is access function (D : in Real) return Real;                   -- 9 Corresponding access type
   Result, Value :  Real := 0.0;                                                -- 10
    procedure Process (Behavior : Compute; Input  : in  Real;                   -- 11 Note first parameter type
                                  Output : out Real) is                         -- 12
   begin                                                                        -- 13
        Output := Behavior(Input);                                              -- 14 Reference to a function
   end Process;                                                                 -- 15
begin                                                                           -- 16
   loop                                                                         -- 17
      Text_IO.New_Line;                                                         -- 18
      Text_IO.Put("Enter Real Value (0 to exit): ");                            -- 19
      FIO.Get(Value);                                                           -- 20
      exit when Value = 0.0;                                                    -- 21
      Process(Behavior => Method'Access, Input => Value, Output => Result);     -- 22 Key statement in example
      Text_IO.New_Line;                                                         -- 23
      Text_IO.Put_Line("The result is ");                                       -- 24
   FIO.Put(Result, Fore => 4, Aft => 3, Exp => 0);                              -- 25
   Text_IO.New_Line;                                                            -- 26
  end loop;                                                                     -- 27
end Function_Access_Type;                                                       -- 28
```

5.4.2.2 A Package Example

Many newcomers to Ada find the accessibility rules frustrating when trying to implement access to subprogram solutions across packages. The accessibility rule remains the same, but one must design a bit more carefully to ensure that access types are at the same level (have the same lifetime) as their access objects and vice versa. Here is an example of how to make that work.

We have a package specification in which we declare a set of access types.

```
package Reference_Types is                                          -- 1
   type Int_32 is range -2**31..2**31 - 1;                          -- 2 a signed integer with range
   for  Int_32'Size use 32;                                         -- 3 use 32 bits for the integer
   type Data_Set is array (Natural range <>) of Int_32;             -- 4 unconstrained array of int_32
   type Data_Set_Reference is access all Data_Set;                  -- 5 pointer type to the array type
   type Validate_Routine is access function(Data : Int_32)          -- 6 access type that points to a
                                  return Boolean;                   -- 7  function; access to function
   type Process_Method is access Procedure(Data : Int_32);          -- 8 access type points to
   procedure Process  (Data : in out Data_Set;                      -- 9                    procedure
                       Method : in Process_Method);                 -- 10
   function  Validate (Data : access Data_Set;                      -- 11 access parameter;  in mode
                       Validator : in Validate_Routine) return Boolean;  -- 12 access to function parameter
   function  Validate (Data : in Data_Set;                          -- 13 access parameter;  in mode
                       Validator : in Validate_Routine) return Boolean;  -- 14 access to function parameter
end Reference_Types;                                                -- 15
```

We have a few new ideas in this package. On line 2 we define an signed integer type with a range that can be represented in thirty-two bits. On line 3 we force the representation to thirty-two bits using the 'Size clause. See the Annex K attributes for the definition of this clause. On lines 6 through 8 we declare some access to subprogram types which for parameters in lines 9 through 15. The following package contains declarations for functions for our final example. It depends on package Reference_Types.

```
with Reference_Types;                                            -- 1
package Reference_Functions is                                   -- 2
   function  My_Process return Reference_Types.Process_Method;   -- 3
   function  My_Validator return Reference_Types.Validate_Routine;  -- 4
end Reference_Functions;                                         -- 5
```

Implementation for both packages will be presented a little later. Here is a little test procedure.

```
with Reference_Types;                                              -- 1
with Reference_Functions;                                          -- 2
with  Ada.Text_IO;                                                 -- 3
procedure Test_Reference_Types is                                  -- 4
   Test_Data : Reference_Types.Int_32 := 42;                       -- 5
   package Int_32_IO is new Ada.Text_IO.                           -- 6
           Integer_IO(Num => Reference_Types.Int_32);              -- 7
   Test_Data_Set : Reference_Types.Data_Set(0..20)                 -- 8
                   := (others => Test_Data);;                      -- 9
begin                                                              -- 10
 Reference_Types.Process (Data   => Test_Data_Set,                 -- 11
                          Method => Reference_Functions.My_Process);  -- 12
end Test_Reference_Types;                                          -- 13
```

Line 6 simply demonstrates an instantiation of an I/O package for the Int_32 type. Line 11 calls the procedure, Process from Reference_Types and gives it an actual parameter of My_Process from the package containing the Reference_Functions. So far, everything is at the same level of accessibility. Here are the package bodies for Reference_Types and Reference_Functions.

```
package body Reference_Types is                                         -- 1
   procedure Process  (Data : in out Data_Set;                          -- 2
                       Method : in Process_Method) is                   -- 3
   begin                                                                -- 4
         for I in Data'Range                                            -- 5
      loop                                                              -- 6
         Method(Data(I));                                               -- 7
      end loop;                                                         -- 8
   end Process;                                                         -- 9
   function  Validate (Data : access Data_Set;                          -- 10
                    Validator : in Validate_Routine) return Boolean is  -- 11
   begin                                                                -- 12
      return Validate(Data.all, Validator);                             -- 13
   end Validate;                                                        -- 14
                                                                        -- 15
   function  Validate (Data : in Data_Set;                              -- 16
                     Validator : in Validate_Routine) return Boolean is -- 17
         Without_Error : Boolean := True;                               -- 18
   begin                                                                -- 19
      for I in Data'Range                                               -- 20
      loop                                                              -- 21
         Without_Error := Validator(Data => Data(I));                   -- 22
         exit when not Without_Error;                                   -- 23
      end loop;                                                         -- 24
      return Without_Error;                                             -- 25
   end Validate;                                                        -- 26
end Reference_Types;                                                    -- 27
```

```
package body Reference_Functions is                                             -- 1
   procedure My_Process (Data : Reference_Types.Int_32) is                      -- 2
   begin                                                                        -- 3
      null;                                                                     -- 4
   end My_Process;                                                              -- 5
   function My_Validator (Data : Reference_Types.Int_32) return Boolean is      -- 6
   begin                                                                        -- 7
      return True;                                                              -- 8
   end My_Validator;                                                            -- 9
   function My_Process return Reference_Types.Process_Method is                 -- 10
         Test_Process    : Reference_Types.Process_Method := My_Process'Access;           -- 11
   begin                                                                        -- 12
      eturn Test_Process;                                                       -- 13
   end My_Process;                                                              -- 14
   function My_Validator return Reference_Types.Validate_Routine is             -- 15
         Test_Validation : Reference_Types.Validate_Routine                     -- 16
           := My_Validator'Access;                                              -- 17
   begin                                                                        -- 18
      return Test_Validation;                                                   -- 19
   end My_Validator;                                                            -- 20
end Reference_Functions;                                                        -- 21
```

Study these to determine where the 'Access attribute is applied. Note how this can actually work and still prevent the dangling references. Accessibility rules are there to keep you from making stupid errors.

6. Subprograms

procedures and functions

Subprograms are either functions or procedures. A subprogram may have parameters or not. Subprogram parameters were introduced in an earlier section. The algorithmic code in your program will almost always be contained within some kind of subprogram (or a task). A subprogram may have locally declared variables, locally declared types, and locally nested subprograms or packages.

6.1 Procedures

6.1.1 Procedure Format and Syntax

A procedure in Ada may be used to implement algorithms. As shown earlier, procedure have a rich set of parameter types and parameter modes. The format of a procedure body is,

```
procedure  Ahoy_There  is                        --  1 Procedure declaration with no parameters;    6.3
   -- procedure declarations                     --  2 Local to this procedure
begin                                            --  3 Begins sequence of algorithmic statements;   6.3
    -- handled sequence of statements            --  4 Handled by exception handler on error         A.10.6
exception                                        --  5 An optional exception handler for the procedure
    -- a sequence of statements handling the exception  --  6 Any handling statements legal
end Ahoy_There ;                                 --  4 Scope terminator with name of unit            6.3
```

6.1.2 Procedure Compilation Units

Note the four parts to the procedure. This is sometimes called the "Ada comb." You may compile a procedure specification as a source file separately from its implementation.

```
with Ada.Text_IO;                          --  1 Put Text_IO library unit in scope;          10.1.2, A.10
procedure  Simple_2;                       --  2 Specification for a procedure may be compiled     6.3
```

The implementation may be coded and compiled later. The implementation for Simple_2 could be,

```
procedure  Simple_2  is                    --  1 Parameterless declaration;  6.3
begin                                      --  2 Begins sequence of algorithmic statements;           6.3
    Ada.Text_IO.Put_Line(“Hello Ada”);     --  3 Dot notation makes Put_Line visible      A.10.6
end Simple_2 ;                             --  4 Scope terminator with name of unit       6.3
```

Another version of this might execute the Put_Line some given number of times using a ***for loop***. A ***for loop*** includes an index value declared locally to the loop and a range of values for the index. The loop will then iterate the number of times indicated by the index range. For example,

```
with Ada.Text_IO;                          --  1 Put Text_IO library unit in scope;          10.1.2, A.10
procedure  Simple_2  is                    --  2 Parameterless declaration;  6.3
begin                                      --  3 Begins sequence of algorithmic statements;           6.3
    for Index in 1..10  loop               --  4 Specification of a for loop
        Ada.Text_IO.Put_Line(“Hello Ada”); --  5 Dot notation makes Put_Line visible      A.10.6
    end  loop;                             --  6 End of loop scope.  End of loop index scope
end Simple_2 ;                             --  7 Scope terminator with name of unit       6.3
```

A variation on the previous program uses some local declarations, a function with a parameter and a simple call from the main part of the procedure.

```
with Ada.Text_IO;                          --  1  Put Ada.Text_IO Library Unit in scope
procedure  Simple_2  is                    --  2 Declaration for parameterless procedure
    function Is_Valid (S : String)         --  3 Declaration for a function with a parameter
        return Boolean is                  --  4 Specify the type of the return value
```

```
        ...                                   --  5 three dots is not legal Ada
     end Is_Valid;                            --  6 End of function scope
     Text : String (1..80);                   --  7 Declare a String variable with constraint
     Len  : Natural;                          --  8 Uninitialized variable
begin                                         --  9  Begin handled-sequence-of-statments
     Ada.Text_IO.Get_Line(Text, Len);         -- 10 Call to Get_Line procedure with two parameters
     if Is_Valid(Text(1..Len)) then           -- 11 Call the function with string parameter
          Text_IO.Put_Line(Text(1..Len));     -- 12 Put string w/carriage return and line feed
     end if;                                  -- 13 Ends scope of if statement
end Simple_2 ;                                -- 14 Ends scope of Simple_2
```

6.1.3 A Simple Main Procedure

A main procedure is not required in Ada 95. However, most of your programs will have one. Here is an example of such a procedure.

```
with Application;  -- This could be any Application package        --  1 Put package Application in scope;  10.1.2,
procedure  Main  is                                                --  2 Parameterless declaration;              6.3
   The_Application : Application.Application_Type;                 --  3 Some kind of  type for the application
begin    -- Main                                                   --  4 Begins Main subprogram;  6.3
     Restart_Iterator:                                             --  5 We want a non-stop system so we
     loop                                                          --  6 create a restart iterator  as a loop.
          Application_Control:                                     --  7 Label the Application control block
          begin    -- Application_Control                          --  8 No harm in commenting every begin
               Application.Start(Data => The_Application);         --  9 Start the application code
               Application.Stop(Data => The_Application);          -- 10 Stop the application code
               exit Restart_Iterator;                              -- 11 If all goes well, exit the loop here.
          exception                                                -- 12 If there is an exception anywhere, do this.
               when  others =>                                     -- 13 Others captures any kind of exception
                    Application.Cleanup(Data => The_Application);  -- 14 Start the cleanup before Restarting
                    Application.Restart  (Data => The_Application); -- 15 Now restart the application
          end Application_Control;                                 -- 16 Block label required because it is labeled
     end loop Restart_Iterator;                                    -- 17 Loop label required because it is labeled
     Application.Finalization (Data => The_Application);           -- 18 The finalization routines for application
end Main;                                                          -- 19 Scope terminator with unit name  6.3
```

6.1.4 Procedure Parameters

Any procedure or function may have parameters. The following example is a variation on the Diamond procedure and demonstrates the use of named association in calling formal parameters. The syntax for named association is (*formal-parameter-name => actual-parameter-name*). This example was originally designed and programmed by a young US Marine Corps Lance Corporal who, at the time, had a high-school education. Notice that he used his knowledge of elementary algebra to write this program with only one loop and simply called the inner procedure by changing the algebraic signs of the actual parameters. While one can easily find ways to improve on this code, it demonstrates how this young Marine thought about the problem before coding it.

```
-- ===============================================            --  1  These first five lines illustrate a useful
-- diamond.ada                                                 --  2  technique for documenting Ada source
-- Solution to Diamond Problem by LCPL Mathiowetz, USMC        --  3  code unit.   The author of this solution
-- Camp Kinser, Okinawa.  June 1993.  AdaWorks Intro to Ada Class --  4  was a USMC Lance Corporal with a
-- ===============================================            --  5  High School education. Very bright man.
with ada.text_io; use Ada;  -- Makes all of package Ada visible --  6 Only Text_IO is required for this program
procedure Diamond is                                           --  7  Specification with no parameters
     package TIO renames Text_IO;                              --  8  A shortened name for Text_IO
     subtype Column is TIO.Positive_Count;                     --  9  Subtype may be used with its parent type
     Center : constant := 37;                                  -- 10 A named constant
     Left_Temp, Right_Temp : Integer := Center;                -- 11 Temporary values, initialized
     Plus_2     : constant :=  2;                              -- 12 Positve constant value
     Minus_2 : constant := -2;                                 -- 13 Negative constant value
```

```
     procedure Draw (Left, Right, Depth : in Integer) is         -- 14 Nested procedure with parameter list
         Symbol : String(1..1) := "X";                           -- 15 The character we will print
         Left_Col, Right_Col : Column;                           -- 16 These are probably extraneous
     begin                                                       -- 17 We are in a nested procedure
      for Index in 1..Depth loop                                 -- 18  Index declared here; type is range type
        if Left_Temp = Center then                               -- 19 Is it time to Put the center character?
          TIO.Set_Col(Center);                                   -- 20 Using renamed Text_IO.Count
          TIO.Put(Symbol);                                       -- 21
        else                                                     -- 22
          Left_Col  := Column(Left_Temp);                        -- 23 Extraneous assignment on these two lines;
          Right_Col := Column(Right_Temp);                       -- 24 we could do type conversion in Set_Col
          TIO.Set_Col(Left_Col);                                 -- 25 TIO.Set_Col(Column(Right_Temp))
          TIO.Put(Symbol);                                       -- 26 might be better coding on line 25 and 27
          TIO.Set_Col(Right_Col);                                -- 27
          TIO.Put(Symbol);                                       -- 28
        end if;                                                  -- 29
        TIO.New_Line;                                            -- 30
        Left_Temp  := Left_Temp  + Left;                         -- 31 Arithmetic on Temporary values using
        Right_Temp := Right_Temp + Right;                        -- 32 algebraic addition on negative parameter
      end loop;                                                  -- 33
   end Draw;                                                     -- 34  End of nested procedure
begin -- Diamond                                                 -- 35  Always comment this kind of thing
   Draw (Left => Minus_2, Right => Plus_2,  Depth =>  9);        -- 36 Use named association for these calls.
   Draw (Left => Plus_2,  Right => Minus_2, Depth => 10);        -- 37 Reverse the signs to get a different shape
end Diamond;                                                     -- 38 End of unit with named unit at end
```

Sometimes we want a variable to enter the procedure with one value and exit with a new value. Here is a simple procedure which uses ***in out*** parameter mode. Although this example is trivially simple, it can be extended to a large range of other data types where one must alter that state of an object in some carefully controlled way.

```
procedure Update (Data : in  out Integer) is                     -- 1  in out allowed on either side of :=
begin                                                            -- 2  start algorithmic part of procedure
    Data := Data + 1;                                            -- 3  In with one value; out with a new value
end Update;                                                      -- 4  end of unit with unit name
```

Other times, it is useful to get a variable with an in value and return some other value within a procedure parameter list. This is not always a good design model since it leads us to combine two ideas, modifier and query, into a single operation. Many OOP practitioners suggest that modifiers and queries should be kept separate. This example shows an update operation on an AVL Tree in which the procedure returns a Boolean to indicate whether the tree is now in balance.

```
procedure Balance (The_Tree : in out AVL_Tree;  Balanced : out Boolean) is   -- 1 Dynamically, self-balancing tree
begin                                                                         -- 2  built on access types for flexibility.
    -- long, complex, dynamically self-balancing algorithm                    -- 3 node rotations: LL, LR, RR, RL
    Balanced :=  -- a boolean result from the balancing algorithm             -- 4 Must be checked by caller
end Balance;                                                                  -- 5
```

The problem with the above example is that, any subprogram making the call, must also be sure to check the Boolean result. If the *Balanced* parameter is not evaluated, the Boolean out parameter is of no value.

```
procedure Insert (Tree : in  out AVL_Tree;  Value : in Item) is               -- 1 From collection of AVL_Tree methods
    OK_To_Proceed : Boolean := False;                                         -- 2 Should be initialized
begin   -- Insert                                                             -- 3 Good practice to comment a begin
    -- algorithm to insert a node in the tree                                 -- 4 Pre-order, in-order, post-order?
    Balance(The_Tree => Tree, Balanced => OK_To_Proceed);                     -- 5 Named association call
    if OK_To_Proceed then                                                     -- 6 If you fail to do this check, you are
         -- some additional source code here                                  -- 7 Making use of  the out parameter of
    end if;                                                                   -- 8 type Boolean.
end Insert;                                                                   -- 9 If  name is supplied, compiler checks.
```

Some Ada practitioners believe it is better to raise an ***exception*** in a function than to return a Boolean ***out*** parameter in a procedure. Their rationale for this is that an ***exception*** cannot be ignored, but an ***out*** parameter, is easy to overlook or ignore.

6.2 Functions

A function must return a result of the type indicated in its profile. The compiler will check for this and not permit any errors. A function may be called as part of an assignment statement or as an argument returning a type within another function or procedure call. Ada also allows pointers (access types) to reference functions.

6.2.1 Function Format and Design

The Is_Valid function from a previous section might be coded to look like this,

```
function Is_Valid (S : String)                   --  1 Default mode is in for type String
                 return Boolean is               --  2 Boolean defined in package Standard
   Result : Boolean := True;                     --  3 Return type named Result as local variable
begin                                            --  4 Begin the handled-sequence of statements
   for I in S'Range loop                         --  5 I takes the index type of String:  Positive
      case S(I) is                               --  6 Examine a single character from the String
         when 'a'..'z' | 'A'..'Z' =>             --  7 Check both upper and lower case
            null;                                --  8 No break statement is required
         when others =>                          --  9 others required if not all options are covered
            Result := False;                     -- 10 Simple assignment of Boolean value
            exit;                                -- 11 exit leaves the loop.  all indices are reset
      end case;                                  -- 12 Every control structure requires terminator
   end loop;                                     -- 13 Ends the scope of the loop including, I
   return Result;                                -- 14 Compiler requires a return statement
end Is_Valid;                                    -- 15 Scope terminator for the function. Required.
```

6.2.2 Function Examples

The next program is an example of an Ada function. This function simply evaluates the greater of two values in a parameter list and returns it. Every function must have at least one return statement.

```
function  Largest (L, R : Integer) return Integer is   -- 1 Parameterized function declaration;   6.3
begin                                                  -- 2 Begins sequence of algorithmic statements;        6.3
   if  L > R then                                      -- 3 Compare L to R
      return L;                                        -- 4 function must return a value of return type        6.3
   else                                                -- 5 If the comparison is false              5.3
      return R;                                        -- 6 Another return;  would a single return be better?
   end if;                                             -- 7 Every if must have a corresponding end if.          5.3
end Largest;                                           -- 8 Scope terminator with name of unit     6.3
```

To call this function you will use an assignment statement.

```
with Largest;                                               -- 1 with is permitted for library unit function
procedure Hrothgar (Y, Z :  in  Integer; X : out Integer) is  -- 2  Note the modes of the parameter list
begin                                                       -- 3
   X := Largest(L => Y,  R => Z);                           -- 4 Named Association syntax                     6.3
end Hrothgar;                                               -- 5 As usual, include the name with the end statement
```

Line 4 shows *named association* syntax. In this case, L and R name the formal parameters. Y and Z name the actual parameters. The arrow, in the form of =>, associates the actual paramter with the formal. This is a powerful feature, unique to Ada, that makes source code more readable and more maintainable.

Suppose we have a record type called Stack. It contains two components. Every ***type*** ... ***is record*** declaration must contain an ***end record*** statement. In the Stack record, shown below, there is also a component of an array type. This is a constrained array of type Stack_Data.

```
type Stack_Data is array(1..1000) of Integer;       -- 1  Constrained array type definition for Integers
type Stack is record                                -- 2  Record type format
   Data : Stack_Data;                               -- 3  Array component within a record
   Top  : Natural := 0;                             -- 4  Natural data type;  note the initialization
end record;                                         -- 5  Every record structure requires an end record
```

Here is a function that returns a boolean value for a record type, Stack, that contains a component, Top

```
function  Is_Empty (S : Stack) return Boolean  is   -- 1  Parameterized function declaration;   6.3
   Result : Boolean := False;                       -- 2  A locally declared result variable
begin                                               -- 3  Begins sequence of algorithmic statements;       6.3
   if  S.Top = 0 then  --  Equality test            -- 4  Syntax for an if statement; then is required
      Result := True;                               -- 5  Assignment statement based on true path
   else                                             -- 6  An else takes the false path
      Result := False;                              -- 7  Another assignment
   end if;                                          -- 8  An if requires an end if;  checked by compiler
   return  Result;                                  -- 9  A function must contain at least one return
end Is_Empty;                                       -- 10 Scope terminator with name of unit    6.3
```

Would it be better to have coded the Is_Empty function as,

```
function  Is_Empty (S : Stack) return Boolean  is   -- 1 Parameterized function declaration;    6.3
begin                                               -- 2 Begins sequence of algorithmic statements; 6.3
   return  S.Top =  0;                              -- 3 Compare S.Top to Zero True or False
end Is_Empty;                                       -- 4 Scope terminator with name of unit     6.3
```

Function parameters are only allowed to be ***in*** or ***access*** mode. The default mode is always in. An ***in*** parameter is the equivalent of a ***constant*** to the function. That is, you can never assign a value to an ***in*** mode parameter value. For an enumerated type, Month, where you want to cycle through the months, returning to January when you reach December. Consider,

```
type Month is (January, February, March, April, May, June, July, August, September, October, November, December);

function Next (Value : Month) return Month  is      -- 1 Declare a parameterized function
begin                                               -- 2 No other declarations
   if  Value = Month'Last then                      -- 3 Month'Last is December
      return Month'First;                           -- 4 Month'First is January
   else                                             -- 5 The usual behavior of else
      return Month'Succ(Value);                     -- 6 Month'Succ(June) is July
   end if;                                          -- 7 End Scope of if statement
end Next;                                           -- 8 End scope of function
```

Consider another type, Vector, defined as an unconstrained array:

```
type Vector  is array (Positive range <>) of Float;    -- An unconstrained array;  must be constrained when used
```

with an exception defined in a visible package specification as:

```
Range_Imbalance : exception;      -- An exception declaration, visible somewhere in the design
                                  -- Note:  an exception is not a data type

function  "+"  (L, R : Vector) return Vector  is        -- 1 Overloading an infix operator
   Result : Vector (L'Range) := (others => 0.0);        -- 2 Constrain and initialize the result array
begin                                                   -- 3
   if L'Length  /= R'Length then                        -- 4 Ensure R and L are of the same length
      raise Range_Imbalance;                            -- 5 Raise user-defined exception shown above.
   end if;                                              -- 6 We never reach this point if exception is raised
```

```
      for Index  in L'Range                      --  7  The 'Range attribute generalizes the Index
          loop                                   --  8  Index only lives the scope of the loop
              Result (Index) := L(Index) + R(Index);  --  9  Index is a constant in the loop
          end loop;                              -- 10 The end of scope for the loop
      return Result;                             -- 11 No exception handler.  The exception is propogated
  end  "+";                                      -- 12                 to the calling subprogram.  Looks for handler.
```

If the exception is not handled locally, the RTE will unwind through the calling stack searching for a handler. If none is found, the program will *crash and burn*. You might want to have a function with an access parameter. This has potential side effects. Consider the following record definition,

```
      type Data is record                  --  1 Define a record type with a name
          Value : Integer := 0;            --  2 Initialize the values when possible
          Description : String(1..20);     --  3 Probably should be initialized
      end record;                          --  4 Scope terminator for the record data
      type Ref is access all Data;         --  5 Define a  pointer to the record
```

You could have a function,

```
  function  Is_Zero  (The_Data : access Data) return Boolean  is     -- 1  Note access parameter
  begin                                                              -- 2  Of course, by now you know this
     return The_Data.Value = 0;                                      -- 3  Return result of equality test
  end Is_Zero;                                                       -- 4  Scope terminator for the function
```

It is not possible to do the following,

```
  function  Fix_It_A  (The_Data : access Data) return Ref  is        --  1 Access parameter and access result
       Fix_It_Data : Ref := new Data'(some initial values);          --  2  Declare some initialized access object
  begin                                                              --  3  Of course, by now you know this
     The_Data := Fix_It_Data;  -- illegal, illegal, illegal          --  4  No assignment allowed to parameter value
     return The_Data;                                                --  5  Will never get to this; will not compile
  end Fix_It_A;                                                      --  6  Scope terminator for the function
```

but is permitted to do this, unfortunately,

```
  function  Fix_It_B  (The_Data : access Data) return Ref  is        --  1 Access parameter and access result
       Fix_It_Data : Integer := 25;                                  --  2  Declare initialized Integer  object
  begin                                                              --  3
     The_Data.Value := Fix_It_Data;                                  --  4  Assignment allowed to component
     return The_Data;                                                --  5  Yes. Returns updated value for The_Data
  end Fix_It_B;                                                      --  6  Always include the name of the function
```

This is one of Ada's weaknesses vis a vis C++. In C++ we can declare a function as *const* or a parameter as *const*. This may be strengthened in a future ISO Ada standard so the access parameter can be ***constant***.

One of the useful algorithmic capabilities of modern programming languages is *recursion*. For a recursive solution, the subprogram must include a way to terminate before it runs out of memory. The following academic example for a recursive function, is seldom a practical in real progamming applications.

```
  function Factorial (N : Natural )          --  1
            return Positive is               --  2 Must have a return type
  begin                                      --  3 Start of algorithmic part
     if  N <= 1 then                         --  4 Less than or equal to ...
         return 1;                           --  5 Lowest positive value
     else                                    --  6 Alternative path
         return N * Factorial (N - 1);       --  7 The recursive call; function calls itself
     end if;                                 --  8 Terminate if statement
  end Factorial;                             --  9 Scope of the recursive function
```

Many sort routines, tree searching routines, and other algorithms use recursion. It is possible to do this in Ada because every subprogram call is re-entrant. Each internal call of itself puts a result in a *stack frame*.

When the algorithm reaches a stopping point, based on the if statement, it unwinds itself from the stack frame entries with a final result of the computation. The following program will work to test the Factorial program,

```
with Factorial;                                   --  1 Yes, you may with a subprogram
with Ada.Integer_Text_IO;                         --  2 I/O for Standard Integer
with Ada.Text_IO;                                 --  3 Character and String I/O
use  Ada;                                         --  4 Make Ada visible; not a problem
procedure Test_Factorial is                       --  5 Specification with "is"
    Data : Natural := 0;                          --  6 In scope up to end of procedure
begin                                             --  7 You know what this means by now
    Text_IO.Put("Enter Positive Integer: ");      --  8 Display a prompt on the screen
    Integer_Text_IO.Get(Data);                    --  9 Get an integer from the keyboard
    Integer_Text_IO.Put(Factorial(Data));         -- 10 Display an integer on the screen
end Test_Factorial;                               -- 11 End of declarative region for procedure
```

Note: Although this is the usual example given in textbooks to illustrate recursion, it is not always the best way to accomplish factorial computation.

It is important to understand that recusion can result in a Storage_Error (see package Standard). Also, intelligent use of Ada's visibility rules can often prevent accidental, infinite recursion.

A function can be compiled by itself in the library. Even more interesting is that a function specification can be compiled into the library by itself. When the specification is compiled it must be completed later with an implementation. This is identical to the procedure example, Simple_2, in 6.1.2 above.

6.3 Subprograms in A Package

An Ada package specification may group a set of subprogram declarations. No implementation code is permitted in the specification. The implementation will be in the package body. This is more fully covered in Chapter 7, below. Here is a simple package specification with a corresponding body. First the specification:

```
package Kia_Ora is                                --  1 Hello in Maori,early language of New Zealand
   procedure Kia_Menemene;                        --  2 Be happy, in Maori
   function   Menemene return Boolean;            --  3 Are you happy?
end Kia_Ora;                                      --  4 end of pacakge specification
```

Then a package body highlighting separate compilation:

```
package body Kia_Ora is                           --  1 Now includes the word, body
   procedure Kia_Menemene is separate;            --  2 Defer actual implementation for the subprograms
   function   Menemene return Boolean is separate; --  3 to separate  compilation units.
end Kia_Ora;                                      --  4
```

The separately compiled procedure could be coded:

```
separate (Kia_Ora)                                --  1 Note absence of semicolon
procedure Kia_Menemene is                         --  2 Makes maintenance much easier in small chunks
begin                                             --  3
   -- some implementation code here               --  4 Any standard Ada algorithmic code here
end Kia_Menemene;
```

7. Package Design

At the beginning of this book, we showed an example of an Ada package. Most Ada programs are designed with packages. In fact, a single program is usually composed of many packages. A *package* is a *module* for collecting related information and services. It can be thought of as a *contract* for services. The user of that contract may be thought of as a *client*. In this sense, a client may us some of the services but not want to use all of those services. Ada allows a client to indentify only those services needed, through its visibility rules, even though all services might be in scope and potentially visible.

The services are in the form of type definitions, data declarations, and subprograms. A well-designed package will rarely have data declarations as part of the contract. Instead, references to data should be through a call to some subprogram.

7.1 A Simple Package

We revise the specification for the earlier Messenger package.

```
package Messenger is                                       --  1 An Ada Module
   type Message is private;                                --  2 A partial definition of message
   function Null_Message return Message;                   --  3 Gives a null message
   function Create (S : String) return Message;            --  4 Make a message from a String
   function Get return Message;                            --  5 Get  message from keyboard      Public Part
   procedure Put (M : in Message);                         --  6 Put Message to Screen
   procedure Clear (M : in out Message);                   --  7 Set message to null message
   function Text (M : Message) return String;              --  8 The string portion of  message
   function Length (M : Message) return Natural;           --  9 How many of characters
private                                                    -- 10 Begin private part of package
   type Message is record                                  -- 11 Full definition of message
      Data : String(1..200) := (others => ' ');            -- 12 Message content; initialized   Private Part
      Len  : Natural := 0;                                 -- 13 Message size; initialized
   end record;                                             -- 14 End of message definition
end Messenger;                                             -- 15 End of the specification
```

Notice there is no algorithmic code in a package specification. Ada lets you declare all the subprograms in the specification. The implementation is in another compilation unit called the package body but the specification and body are both part of the same library unit. The specification is a contract with a client. It tells what it will do, not how it will be done. Ada is forbids algorithmic code in the specification part.

A client of package Messenger is only able to see lines 1 through 9 of the specification. The rest (lines 10 through 14) is only in the specification to satisfy the requirements of the Ada compiler. We call lines 1 through 9 the public part of the specification and lines 10 through 14, the private part. The private part of an Ada package specification is somewhat analogous to a C++ class protected part. A child library unit may have some visibility to private part just as C++ derived class has visibility to a protected part of its parent class. We examine these visibility issues later.

The package Messenger exports some services as subprograms. The algorithmic (procedural) part of these subprograms must be coded someplace. Ada forbids algorithms in the package specification. Algorithms must be coded in the package body. Subprogram declarations in the specification require a corresponding implementation in the body. The package body depends on successful compilation of its fully conforming package specification. The Ada compiler checks this dependency through compilation unit date and time stamps. The package body is an integral part of the library unit. The package body never needs to ***with*** the package specification because both are part of the same library unit.

7.2 Package Body

Not every package needs a package body. In practice, only packages that declare public subprograms need a body. Now and then a package may require a body even if it does not export a subprogram. This would be the exception rather than the rule. This exception to the rule is also rigorously managed by the compiler.

Here is a package body for Messenger.

```
package body Messenger is                                              -- 1
   function Create (S : String) return Message is                      -- 2
   begin                                                               -- 3
         -- algorithm to create object of type Message                 -- 4
         -- must have at least one return statement                    -- 5
   end Create;                                                         -- 6
   function Get return Message is                                      -- 7
   begin                                                               -- 8
         -- algorithm to Get a message from some container or input device  -- 9
         -- must have at least one return statement                    -- 10
   end Create;                                                         -- 11
   procedure Put (M : in Message) is                                   -- 12
   begin                                                               -- 13
         -- algorithm                                                  -- 14
   end Put;                                                            -- 15
   procedure Clear (M : in out Message) is                             -- 16
   begin                                                               -- 17
         -- algorithm to clear the Message                             -- 18
   end Clear;                                                          -- 19
   function Text (M : Message) return String is                        -- 20
   begin                                                               -- 21
         -- algorithm, if necessary                                    -- 22
         -- must have at least one return statement                    -- 23
   end Text;                                                           -- 24
   function Length (M : Message) return Natural is                    -- 25
   begin                                                               -- 26
         -- algorithm to get length of Message Text                    -- 27
         -- must have at least one return statement                    -- 28
   end Length;                                                         -- 29
end Messenger;                                                         -- 30
```

An acceptable variation on this body would be to code each subprogram with the reserved word ***separate***. For example,

procedure Put
 (M : **in** Message) **is separate**;

This would cause a stub for a subunit to be created in the library for the completed code corresponding to procedure Put. This technique is useful when one wants to divide the implementation of a package over a team of several people, or preserve the confidentiality of a particular piece of source code.

Neither a client or child of package Messenger ever has visibility to the package body. We say that the implementation (always in a package body) is *encapsulated*.

7.3 More Simple Package Examples

7.3.1 Monetary Conversion Package

Here is another simple package specification. An implementation would convert currencies.

```
package Conversions is                                        -- 1
    type Money is digits 12 delta 0.0001;                     -- 2 a decimal fixed-point type
    type Yen is new Money;                                    -- 3 derive from Money
    type Dollars is new Money;                                -- 4 derive from Money
    function Convert (Y : Yen; Rate : Money) return Dollars;  -- 5 declare a function specification
    function Convert (D : Dollars; Rate : Money) return Yen;  -- 6 declare a function specification
    Conversion_Error : exception;                             -- 7 declare an exception
end Conversions;                                              -- 8
```

```
package body Conversions is                                      -- 1
   function Convert (Y : Yen; Rate : Money) return Dollars is    -- 2
      Result : Dollars := 0.0;                                   -- 3 declare result of return type
   begin                                                         -- 4 stub out the function temporarily
      return Result;                                             -- 5 after algorithm to do conversion
   end Convert;                                                  -- 6
   function Convert (D : Dollars; Rate : Money) return Yen is    -- 7
      Result : Yen := 0.0;                                       -- 8 declare result of return type
   begin                                                         -- 9 temporarily stub out the begin..end part
      return Result;                                             -- 10  after algorithm to do conversion
   end Convert;                                                  -- 11
end Conversions;                                                 -- 12
```

The technique here is to stub out a function. Notice we must first declare a Result of the return type. Then we can code the return statement in the begin..end part. A procedure can be stubbed out with the reserved word, null. A function must have at least one return statement. This technique satisfies that requirement.

7.3.2 Simple Statistics Package

Here is another kind of package. This package provides a simple set of statistical services.

```
package Statistics is                                        -- 1 Specification declaration
   type Data is array (Positive range <>) of Float;          -- 2 An  unconstrained array.
   function Mean     (The_Data : Data) return Float;         -- 3 Computes the statistical Mean
   function Mode     (The_Data : Data) return Float;         -- 4 Computes the statistical Mode
   function Max      (The_Data : Data) return Float;         -- 5 Computes Maximum Value of  arrray
   function Min      (The_Data : Data) return Float;         -- 6 Computes Minimum Value of array
   function Variance (The_Data : Data) return Float;         -- 7 Computes Statistical Variance
   function StdDev   (The_Data : Data) return Float;         -- 8 Computes Standard Deviation
end Statistics;                                              -- 9  Package specification requires end
```

The following procedure is a client of the Statistics package.

```
with Statistics;                                         -- 1  Put Statistics library unit in scope
with Ada.Float_Text_IO;                                  -- 2  Library unit for floating point I/O
use  Ada;                                                -- 3  Makes Ada visible; discussed later
procedure Compute_Statistics is                          -- 4  A stand-alone procedure
   Stat_Data : Statistics.Data(1..100);                  -- 5  An array of float; note the constraint
begin                                                    -- 6  Starts the algorithmic part of procedure
   for Index in Stat_Data'Range                          -- 7  Specification of a for loop; more later
      loop                                               -- 8  Every loop must have the word loop
         Float_Text_IO.Get(Stat_Data(Index));            -- 9  Fill the array with data
      end loop;                                          -- 10 Every loop must have an end loop
   Float_Text_IO.Put(Statistics.Mean(Stat_Data));        -- 11 Call Statistics.Mean and output result
   Float_Text_IO.Put(Statistics.StdDev(Stat_Data));      -- 12 Call Statistics.StdDev and output result
end Compute_Statistics;                                  -- 13 End of the procedure scope
```

The ***with*** statement on Line 1 puts the resources of the Statistics package in scope. The Variance function may be called by referencing Statistics.Variance. Line 2 puts the language-defined library unit, Ada.Float_Text_IO in scope. Line 3 makes the parent of Float_Text_IO directly visible. Therefore, the Get operation of Float_Text_IO on Line 9 is legal. Program declarations are between the ***is*** on Line 4 and the ***begin*** on Line 6. On Line 5, the declaration is for data of the array type Statistics.Data. Since Statistics.Data is declared with no actual range in the Statistics package, the programmer must specify beginning and ending index values. Ada allows starting indexes other than zero. The defined index for an array type may even include a range of negative values.

The expression, Stat_Data'Range in the loop specification, indicates that the loop will traverse the entire array, beginning with the first value through the last value. The loop index, Index, will start with the first value in the Range and proceed to the end. The Get operation on Line 9 is defined in the package Ada.Float_Text_IO. Because we have a use clause for Ada on Line 3, we may reference it as shown.

The same is true for the Put operations on Lines 11 and 12. We call the Mean and StdDev functions from Statistics. These functions take a parameter of type Data and return a floating point value.

7.4 Simple Mathematics Packages

Ada has a rich set of capabilities for numeric algorithms. One of the key packages is Ada.Numerics. This package has some child packages. The most important are Ada.Numerics.Generic_Elementary_Functions, Ada.Numerics.Float_Random, and Ada.Numerics.Discrete_Random. It also defines, in Annex G, a model for *strict* and *relaxed* mode for floating point values.

7.4.1 Example without Numerics Library

You do are not required to use the standard libraries for numerics. This example will compile.

```
with Ada.Text_IO;                   -- 1 Put Text_IO library unit in scope;        10.1.2, A.10
with Ada.Float_Text_IO;             -- 2 Predefined in Annex A                     A.10.9/33
procedure Pi_Symbol is              -- 3 Parameterless declaration;                6.3
   Pi : constant Float := 3.1415;   -- 4 Should have used Ada.Numerics for this
   Radius : Float := 12.0;          -- 5 Ordinary Floating point initialized
   Area : Float := 0.0;             -- 6 I prefer to initialize all variables; not require here
begin                               -- 7 Begins sequence of algorithmic statements; 6.3
   Area := Pi * Radius ** 2;        -- 8 Possible to paste in the special character
   Ada.Float_Text_IO.Put(Area);     -- 4 Dot notation makes Put visible            A.10.6
end Pi_Symbol;                      -- 5 Scope terminator with name of unit        6.3
```

7.4.2 Using Numerics Library

A better approach to declaring Pi and and using Ada for number crunching is to use the language-defined numerics libraries. The following program illustrates some ideas from this set of libraries.

```
with Ada.Text_IO;                                -- 1 Put Text_IO library unit in scope; 10.1.2, A.10
with Ada.Float_Text_IO;                          -- 2 A.10.9/33
with Ada.Numerics.Generic_Elementary_Functions;  -- 3 A.5.1
use Ada;                                         -- 4 Gives direct visibility to all of package Ada   8.4
procedure Compute_Trigs is                       -- 5 Parameterless declaration;          6.3
   package Compute is new Ada.                   -- 6 A.2  A new instance with a new name
                     Numerics.                   -- 7 A.5 Root package for numerics
                     Generic_Elementary_Functions -- 8 A.5.1  Contains Trig and other functions
                     (Float_Type => Float);      -- 9 A.1/25 for definition of type Float
   Pi  : Float := Ada.Numerics.Pi;               -- 10 Pi is defined in Ada.Numerics
   Radius : Float := 12.0;                       -- 11 Ordinary Floating point initialized
   Area : Float := 0.0;                          -- 12 I prefer to initialize variables; not required here
   SQRT_Result : Float := 0.0;                   -- 13 For our Square root computation
begin                                            -- 14 Begins sequence of algorithmic statements;   6.3
   Area := Pi* Radius ** 2;                      -- 15 Compute the area of the circle
   Ada.Float_Text_IO.Put(Area);                  -- 16 dot notation makes Put visible      A.10.6
   Sqrt_Result := Compute.Sqrt(Area);            -- 17 Note use of Compute with dot notation
end Compute_Trigs;                               -- 18 Scope terminator with name of unit            6.3
```

Note: Not everyone agrees with line 12, above. Some developers prefer not to initialize variables because they might contribute to unexpected errors during maintenance.

7.4.3 Precompile Numerics Library

Sometimes it is useful to precompile a generic library package for a frequently used data type. The math library is one such package, especially if you are using the same floating point type over and over in your application.

Consider,

```
package Defined_Types is
      type Real is digits 7 range -2.0 ** 32 .. 2.0 ** 32;
end Defined_Types;
```

Now you could precompile the generic elementary functions package for this type so it could be brought into scope through a simple "with" clause.   For example,

```
with Ada.Numerics.Generic_Elementary_Functions;
with Defined_Types;
package Real_Functions is new Ada.Numerics.
                                              Generic_Elementary_Functions(Defined_Types.Real);
```

This fragment of code can actually be compiled as a new library unit that can be referenced in a context clause through a with clause

Now, you can access this package easily by "**with** Real_Functions" in a context clause.

7.4.4 Mathematical Expressions

The following examples demonstrate the use of the generic mathematics package with calls to some of the functions in that package. Note that the default type for trigonometric functions is in Radians.

```
with Defined_Types;                                                                                   --  1
with Real_Functions;                                                                                  --  2
with Generic_Utilities;                                                                               --  3
procedure Test_Math_Functions is                                                                      --  4
  subtype Degree  is Defined_Types.Real range 1.0..360.0;                                             --  5
  subtype Radian  is Defined_Types.Real range 0.0..2.0 * 3.14;                                        --  6
  function To_Degrees is new Generic_Utilities.To_Degrees(Degree => Degree, Radian => Radian);        --  7
  function To_Radians is new Generic_Utilities.To_Radians(Degree => Degree, Radian => Radian);        --  8
  R1, R2, R3, R4 : Radian := 0.0;                                                                     --  9
  D1 : Degree := 90.0;                                                                                -- 10
  D2 : Degree := 360.0;                                                                               -- 11
begin                                                                                                 -- 12
     R1 := To_Radians(D1);                                                                            -- 13
     R2 := Real_Functions.Sin(X => R1);                                                               -- 14
     R2 := Real_Functions.Sin(X => R1, Cycle => D2);                                                  -- 15
     R2 := Real_Functions.ArcSinh(X => R1);                                                           -- 16
     R3 := Real_Functions.ArcCot(X => R1, Cycle => 40.0);                                             -- 17
     R4 := Real_Functions.Cos(X => R1, Cycle => D2);                                                  -- 18
     R1 := To_Radians(D2);                                                                            -- 19
     R3 := Real_Functions.Tan(X => R1);                                                               -- 20
     D2 := To_Degrees(R2);                                                                            -- 21
end Test_Math_Functions;                                                                              -- 22
```

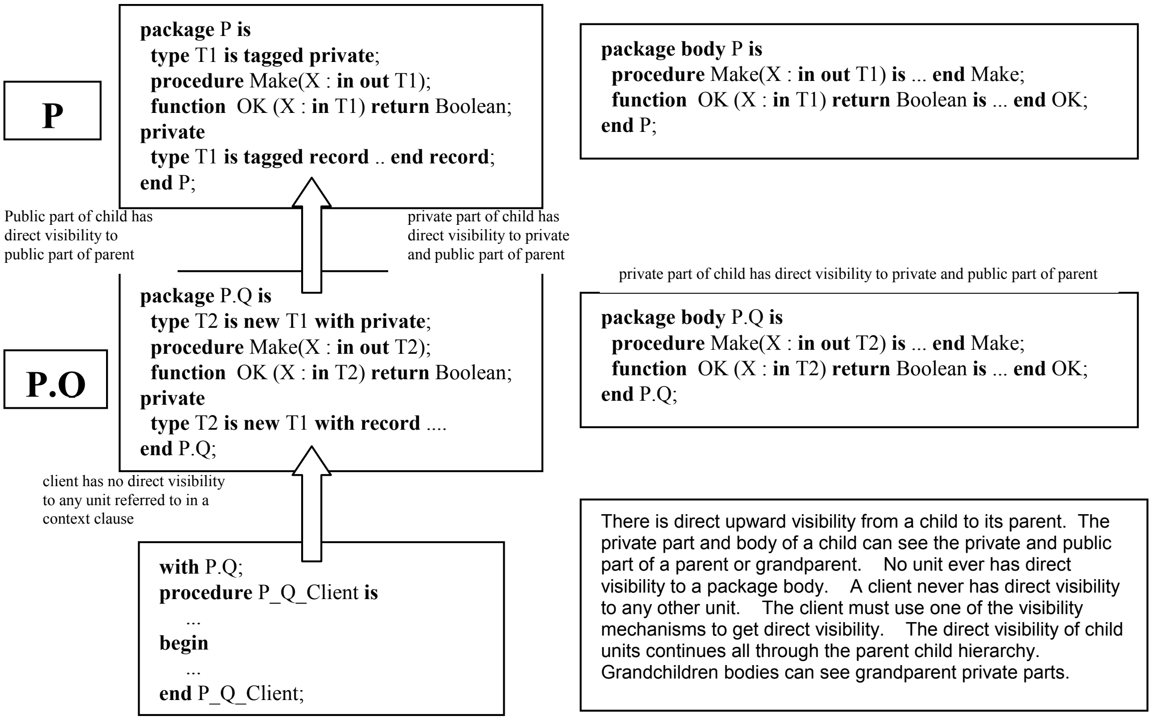
The package Generic_Utilities is not described in this book. It is in the program files that come with this book.   For functions with no cycle parameter, assume a natural cycle of 2 Pi, which means all calculations are done in radians.   Lines 17 shows that you can provide other parameter values for the cycle parameter.

7.4.5 Annex K Attributes

There are a lot of attributes in Annex K specifically designed to enhance your ability to create flexibile, easy to read mathematical expressions.   If you are doing a lot of numerical work, pay particular attention to attributes: Adjacent, Copy_Sign, Denorm, Exponent, Floor, Ceiling, Fraction, Compose, Model, Remainder, Machine_Rounds, Machine_Overflows, other Machine attributes, Rounding, the Safe attributes, Scaling, Signed_Zeros, Unbiased_Rounding, Truncation, all of the Model attributes.   This is not a complete list.   The point of this paragraph is that Ada has a rich set of facilities for numerical analysis and scientific computation.   Also, there are libraries of numerical functions available in public libraries.

8. Child Library Units

An Ada package may have a child. The child may be another package or a subprogram. A subprogram may not have a child. Most of the time, design child library units as packages so they can be extended. A child package specification is just like any other package specification.



```
package P is
  type T1 is tagged private;
  procedure Make(X : in out T1);
  function  OK (X : in T1) return Boolean;
private
  type T1 is tagged record .. end record;
end P;
```

```
package body P is
  procedure Make(X : in out T1) is ... end Make;
  function  OK (X : in T1) return Boolean is ... end OK;
end P;
```

Public part of child has direct visibility to public part of parent

private part of child has direct visibility to private and public part of parent

private part of child has direct visibility to private and public part of parent

```
package P.Q is
  type T2 is new T1 with private;
  procedure Make(X : in out T2);
  function  OK (X : in T2) return Boolean;
private
  type T2 is new T1 with record ....
end P.Q;
```

```
package body P.Q is
  procedure Make(X : in out T2) is ... end Make;
  function  OK (X : in T2) return Boolean is ... end OK;
end P.Q;
```

client has no direct visibility to any unit referred to in a context clause

```
with P.Q;
procedure P_Q_Client is
   ...
begin
   ...
end P_Q_Client;
```

There is direct upward visibility from a child to its parent. The private part and body of a child can see the private and public part of a parent or grandparent. No unit ever has direct visibility to a package body. A client never has direct visibility to any other unit. The client must use one of the visibility mechanisms to get direct visibility. The direct visibility of child units continues all through the parent child hierarchy. Grandchildren bodies can see grandparent private parts.

8.1 Root Packages

Sometimes we want to design a root package that is the home node for a hierarchy or subsystem of other library units. A root package can vary greatly in its form. Here is one possible root package

```
package Root is                                          --  1 Declare a root package specification
     Bad_Bad_Bad : exception;                            --  2 An exception declaration which will be
     No_No_No :    exception;                            --  3 visible throughout the entire hierarchy.
     type Number is private;                             --  4 A partial definition for a type
     function "+" (N  : Number)  return Number;          --  5 Overloading equivalent to i++
     function "-" (N  : Number)  return Number;          --  6 Overloading equivalent to i--
     function Set (To : Integer) return Number;          --  7 Set number  to a value
     function Integer_Is(N : Number) return Integer;     --  8 Convert number to an Integer
private                                                  --  9 Begin the private part of package
  type Number is range -2**31..2**31-1;                  -- 10 Full definition of the private type
end Root;                                                -- 11 End of scope for package specification
```

This package illustrates a possible design for a root package. Not every root package will look like this, but we suggest it as food for thought in creating your own root library units. Here is a simple child package of the preceding Root package.

```
package Root.Application is
     type Application_Type is private;
     procedure Create (A : in out Application_Type);
     function   Is_Empty(A : Application_Type) return Boolean;
     -- more operations
```

```
    private
          type Application_Type is ... ;  -- full definition for type
    end Root.Application;
```

Earlier in this book we had a package that resembled the following,

```
package Abstract_Machinery is                                  --  Package specification; requires body
    type Machine is abstract tagged private;                   -- Specifies the visible part of the data type;
    type Reference is access all Machine'Class;                -- Tagged type should have classwide access
    function Create (Desc : String)                            -- Parameter for Create
                     return Machine'Class;                     -- Tagged return type should be classwide
    procedure Turn_On  (M : in out Machine);                   --  procedure specification
    procedure Turn_Off (M : in out Machine);                   --  procedure specification
    function  Is_On (M : in Machine) return Boolean;           --  function specification
private                                                        --  private part hidden from a client of contract
    type Machine is abstract tagged record                     --  full definition of the publicly declared type
       Turned_On : Boolean := False;                           --  component of the type; OOP attribute
       Description : String(1..120);                           --  Constrained array component
    end record;                                                --  scope terminator for the component
end Abstract_Machinery;                                        --  scope terminator for the specification
```

This is a base package for designing many kinds of machines that can be turned on and off. The data type, Machine, is declared abstract. That means no instances of it are allowed. One could create some child packages for this, combining child library units and inheritance.

```
package Abstract_Machinery.Classwide is                        --  Package specification; requires body
    type FIFO_Container(Size : Positive)                       -- Parameterized type; make it any size
              is limited private;                              -- No assignment for limited type
    procedure Put(CM   : in out FIFO_Container;                -- Put into the next available location
                Data : access Machine'Class);                  -- Any member of class, Machine
    procedure Get(CM   : in out FIFO_Container)                -- Get, destructively, first item
                  Data : access Machine'Class);                -- Any member of Machine'class
private                                                        -- Start hidden part of the package
    type Machine_Data is array                                 -- Define an unconstrained array
           (Positive range <>) of Reference;                   -- The array is pointers to Machine'Class
    type FIFO_Container(Size : Positive) is                    -- Full definition of parameterized type
       record                                                  -- In the format of a record
          Current : Natural;                                   -- What is the current item
          Data    : Machine_Data(1..Size);                     -- Pointer array to Machine derivations
       end record;                                             -- Terminate scope of the record
end Abstract_Machinery.Classwide;                              --  scope terminator for the specification
```

This classwide child package will let you put any object of type Machine'Class into a container. This is quite a handy thing to be able to do. You could have a container of different kinds of machines. This is sometimes called a heterogeneous container.

9. Object-Oriented Programming With Packages

One of the powerful features of Ada is its support for inheritance and dynamic binding, two of the key features of object-oriented programming. Ada accomplishes this through the type model. One type may be derived from another and inherit all the properties of the parent type. In object-oriented programming, straight inheritance is not enough. One must be able to extend the derived type with new operations and components. Ada enables this through the tagged type.

9.1 An Object-Oriented Type

Consider this package containing a tagged type. Every instance of a tagged type contains an internal tag. A tagged type may be extended with additional components.

```
package Machinery is                                          --  1  An Ada Module
   type Machine is tagged  private;                           --  2  A tagged  partial definition of message
   type Reference is access all Machine'Class;                --  3  A classwide access type
   procedure Turn_On (M : in out Machine);                    --  5  Turn on the machine
   procedure Turn_Off (M : in out Machine);                   --  6  Turn off the Machine
   function Is_On  (M : Machine) return Boolean;              --  7 Is the Machine turned on?
private                                                       --  8  Begin private part of package
   type Machine is tagged record                              --  9 Full tagged definition of message
      Is_On : Boolean := False;                               -- 10  Machine content; initialized
   end record;                                                -- 11 End of machine definition
end Machinery;                                                -- 12 End of the package specification
```

9.2 A Possible Client of the Type

A client of package Messenger might be set up as,

```
with Messenger;                                               -- 1 A context clause
procedure Messenger_Processor ... end Messenger_Processor;    -- 2 Three dots are not legal Ada
```

The first line, with Messenger, puts the package named Messenger and all of its services in the declarative region available to Messenger_Processor. Those services can be made visible through a use clause, a use type clause, renaming of the operations, or simple dot notation.

9.3 Inheritance and Extension

The Machinery package specification, with its tagged type, Machine, illustrates some important ideas in Ada. A tagged type may be extended. Therefore, one could have a client package, Rotating_Machinery,

```
with Machinery;                                                    --  1
package Rotating_Machinery is                                      --  2
  type Rotational is new Machinery.Machine with private;           --  3 Inherits Machine methods & data
  procedure Turn_On (M : in out Rotational);                       --  4  Overrides Machinery.Turn_On
  procedure Turn_Off (M : in out Rotational);                      --  5  Overrides Machinery.Turn_Off
  procedure Set_Speed(M : in out Rotational;  S : in Positive);    --  6  New primitive operation
private                                                            --  7
  type Rotational is new Machinery.Machine                         --  8
    with record                                                    --  9
       RPM  :  Natural :=  0;                                      -- 10 New component in derivation
    end record;                                                    -- 11
end Rotating_Machinery;                                            -- 12
```

The Rotating_Machinery package declares a data type that extends the content of the parent type. The type, Rotational now contains two components. It has the one originally included in Machine plus the one we added in the type derivation statement.

9.4 Dynamic Polymorphism

The operations Turn_On, Turn_Off, Is_On, and Set_Speed are called ***primitive operations***. They can be called dynamically, depending on the tag of the object. The following procedure demonstrates one way to do this. Note: the actual procedure to be called cannot be determined until run-time in this example.

```
with Machinery, Rotating_Machinery;                  -- 1  Context clause
with Ada.Integer_Text_IO;                            -- 2  Enables the input of the array index
procedure Dynamic_Binding_Example_1 is               -- 3  Specification for the example procedure
Data : array (1..2) of Machinery.Reference :=        -- 4 Anonymous array of access objects
   (1 => new Machinery.Machine,                      -- 5 Dynamically allocate new Object
    2 => new Rotating_Machinery.Rotational);         -- 6 Dynamically allocate new Object
    Index : Natural range 1..2 := 0;                 -- 7  Use this to index into the array
begin                                                -- 8
    Ada.Integer_Text_IO.Get(Index);                  -- 9  Get the index for the next statement
    Machinery.Turn_On(Data(Index).all);              -- 10 Dynamically call one of the Turn_On methods
end Dynamic_Binding_Example_1;                       -- 11
```

The next example does essentially what the previous example did. However, this example illustrates how to code a classwide procedure. Once again, which version of Turn_On to choose is known only at run-time.

```
with Machinery, Rotating_Machinery;                  -- 1 With both packages; no use clause required
with Ada.Integer_Text_IO;                            -- 2  Enables the input of the array index
procedure Dynamic_Binding_Example_2 is               -- 3  Specification for the example procedure
    Data : array (1..2) of Machinery.Reference :=    -- 4 Anonymous array of access objects
   (1 => new Machinery.Machine,                      -- 5 Dynamically allocate new Object
    2 => new Rotating_Machinery.Rotational);         -- 6 Dynamically allocate new Object
    Index : Natural range 0..2 := 0;                 -- 7  Use this to index into the array
    procedure Start(M : Machine'Class) is            -- 8  Procedure with classwide parameter
    begin                                            -- 9
        Machinery.Turn_On(M);                        -- 10  Turn_On is dynamically determined via the tag
    end Start;                                       -- 11
begin                                                -- 12
    Ada.Integer_Text_IO.Get(Index);                  -- 13 Get the index for the next statement
    Start(M => Data(Index).all));                    -- 14 Call the classwide procedure
end Dynamic_Binding_Example_2;                       -- 15
```

Here is still one more example that illustrates the usefulness of a function that returns a classwide value..

```
with Machinery, Rotating_Machinery;                  -- 1  No use clause is required for this example
with Ada.Integer_Text_IO;                            -- 2  Enables the input of the array index
procedure Dynamic_Binding_Example_3 is               -- 3  Specification for the example procedure
    Index : Natural range 0..2 := 0;                 -- 4  Use this to index into the array
    function Get (The_Index : Natural) return Machine'Class  is  -- 5  Procedure with classwide parameter
        Data : array (1..2) of Machinery.Reference :=   -- 6 Anoymous array of access objects
            (1 => new Machinery.Machine,             -- 7 Dynamically allocate new Object
             2 => new Rotating_Machinery.Rotational);   -- 8 Dynamically allocate new Object
    begin                                            -- 9
        return Data(Index).all));                    -- 10   return the data access by Data(Index)
    end Get;                                         -- 11
begin                                                -- 12
    Ada.Integer_Text_IO.Get(Index);                  -- 13 Get the index for the next statement
    declare                                          -- 14 Start a local declare block
        The_Machine : Machine'Class := Get(Index);   -- 15 Declare and constrain classwide variable
    begin                                            -- 16
        Turn_On(The_Machine);                        -- 17 Call classwide procedure dynamically constrained data
    end;                                             -- 18
end Dynamic_Binding_Example_3;                       -- 19
```

10. Using Standard Libraries

String handling is a simple idea that becomes complicated in some programming environments. In particular, C, C++, and COBOL have made this more difficult than it needs to be. Ada is especially handy for string manipulation. Not only is an Ada string easy to declare and process, the language has predefined libraries (in Annex A) for most of the operations one might want to do on strings, a set of convenient attributes (Annex K) for special functions, and simple methods for converting between strings values and numeric values.

10.1 String Examples

This program illustrates several additional features of the language. Notice the syntax for declaring a **constant**. On line 3, if the string variable is declared with a range constraint, the initializing string must have exactly the same number of characters. On line 4, if there is no range constraint, the index of the first character is 1 and the index of the last character is whatever the character count might be, in this case 9. Line 15 “slides” a string slice from one string into a slice in another string using the assignment operator and parenthetical notation to designate the source and target slices.

```
with Ada.Text_IO;                                 --  1 Put Ada.Text_IO library unit in scope;        10.1.2, A.10
procedure  Bon_Jour  is                           --  2 Parameterless declaration;                    6.3
     Hello : String (1..5) := “Salut”;            --  3 Number of characters must match range;        4.1, A.1/37
     Howdy : String := “Howdy Joe”;               --  4 Compiler determines constraint from string;   2.6, 3.3.1/13
     Bon_Jour : constant String := “Bon Jour”;    --  5 A true constant;  cannot be altered;          3.3.1/5-6
begin                                             --  6 Begins sequence of algorithmic statements;    6.3
     Ada.Text_IO.Put(Hello);                      --  7 Put a string with no carriage return;         A.10.6
     Ada.Text_IO.Set_Col(20);                     --  8 On same line, position cursor at column 20;   A.10.5
     Ada.Text_IO.Put_Line(Hello);                 --  9 Put a string with a carriage return / line feed; A.10.7
     Ada.Text_IO.Put(Howdy);                      -- 10 Puta string with no carriage return;          A.10.7
     Ada.Text_IO.Set_Col (20);                    -- 11 Set the cursor to column 20 / line feed;      A.10.5
     Ada.Text_IO.Put(Howdy);                      -- 12 Put a string with no carriage return / line feed;          A.10.7
     Ada.Text_IO.New_Line(2);                     -- 13 Position cursor to a new line;  double space; A.10.5
     Ada.Text_IO.Put_Line(Bon_Jour);              -- 14 Put a constant to the screen with CR/LF;      A.10.7
     Howdy(7..9) := Bon_Jour(1..3);               -- 15 Slide (assign) one string slice into another; 4.1.2
     Ada.Text_IO.Put_Line (Howdy);                -- 16 Put the modified string with CR/LF;           A.10.7
end Bon_Jour;                                     -- 17 Note the label for the enclosing procedure;   6.3
```

There are better alternatives for String handling in a set of packages in Annex A.4 Here is a simple example of one of the packages. This is easier than string slicing and other low-level code.

10.1.1 Using the Fixed Strings Package

```
with Ada.Text_IO;                                 --  1 Put Ada.Text_IO library unit in scope;           10.1.2, A.10
with Ada.Strings.Fixed;                           --  2 A language defined string package                A.4.1, A.4.3
use  Ada;                                         --  3 Makes all of package Ada visible
procedure  Ni_Hao_Ma  is                          --  4 Hello in Mandarin Chinese                        6.3
     Greeting : String(1..80);                    --  5 80 character string;  String defined in package Standard   ALRM A.1
     Farewell : String(1..120);                   --  6 120 character string
begin                                             --  7 Start sequence of statements
     Ada.Strings.Fixed.Move(Greeting, Farewell);  --  8 Move shorter string to longer string; may also move longer to shorter
end Ni_Hao_Ma;                                    --  9 End of procedure scope.
```

10.1.2 Bounded Strings

It is also possible to do operations on Bounded and Unbounded_Strings. Bounded strings are those with a fixed size at compilation time through a generic instantiation. Unbounded strings are those which can be of any size, mixed size, etc. Many compilers will do automatic garbage collection of unbounded strings. If you want to try these two features of the language, they are defined in Annex A.4 of the Ada Language Reference Manual.

10.1.3 Unbounded Strings

Consider the following program that lets you concatenate data to an unbounded string, convert that string to a standard fixed string, and then print it out to the screen.

```
with Ada.Strings.Unbounded;                                      --  1
with Ada.Text_IO;                                                --  2
use  Ada; use Strings;                                           --  3
procedure Unbounded_String_Demonstration is                      --  4
   Input  : Character := ' ';                                    --  5
   Output : String (1..80) := (others => ' ');                   --  6
   Buffer : Unbounded.Unbounded_String;                          --  7
   Length : Natural;                                             --  8
begin                                                            --  9
   loop                                                          -- 10
      Text_IO.Put("Enter a character: ");                        -- 11
      Text_IO.Get(Input);                                        -- 12
      exit when Input = '~';                                     -- 13
      Unbounded.Append(Source => Buffer, New_Item => Input);     -- 14
   end loop;                                                     -- 15
   Length := Unbounded.Length(Buffer);                           -- 16
   Output(1..Length) := Unbounded.To_String(Buffer);             -- 17
   Text_IO.Put_Line(Output(1..Length));                          -- 18
end Unbounded_String_Demonstration;                              -- 19
```

10.1.4 Other String Operations

There are many other facilities for string handling in Ada. We show here an example from another useful library, package Ada.Characters. Here is a little package that converts lower case letters to upper case.

```
with Ada.Text_IO;                                   -- 1 Put Ada.Text_IO library unit in scope;  10.1.2, A.10
with Ada.Characters.Handling;                       -- 2 Character Handling Operations            A.3.2
use  Ada;                                           -- 3 Makes package Ada visible
procedure  Arirang  is                              -- 4 Famous Korean love song                  6.3
  Data : String := "arirang";                       -- 5  initialized lower case character string
begin                                               -- 6 Start sequence of statements
  Text_IO.Put(Characters.Handling.To_Upper(Data));  -- 7 Convert output to upper case characters and print it
end Arirang;                                        -- 8 End of procedure scope.
```

10.2 Converting Strings to Other Types

Sometimes it is necessary to represent a string value in some other format. Other times we need to convert some other type to a string representation. One could easily write a small generic subprogram to accomplish this. Also, Ada provides an unchecked conversion capability. Unchecked features are seldom used since they circumvent the fundamental philosophy of Ada: every construct should be, by default, safe.

10.2.1 Converting a String to an Scalar Type

The following procedure demonstrates many of the features of the language for converting a string to an integer, a string to a floating point, a string to an unsigned number, and a string to an enumerated value.

```
-- ========== String_To_Scalar_Demonstration ===========
-- String_To_Scalar_Demonstration.adb  by Richard Riehle
-- This program demonstrates several ways to convert a
-- a string to a scalar value.
--
-- ====================================================
with Ada.Text_IO;
with Ada.Integer_Text_IO;
with Ada.Float_Text_IO;
use  Ada;
procedure String_To_Scalar_Demonstration is
  type Spectrum is (Red, Orange, Yellow, Green,
                    Blue, Indigo, Violet);
  type Unsigned is mod 2**8;
  Num   : Integer  := 0;
  FNum  : Float    := 0.0;
  Color : Spectrum := Blue;
  MNum  : Unsigned := 0;
  Text  : String(1..10);
  Text_Integer  : String := "451";
  Text_Float    : String := "360.0";
  Text_Color    : String := "Orange";
  Text_Unsigned : String := "42";
  Integer_Last  : Natural;
  Float_Last    : Natural;
  Spectrum_Last : Natural;
  Modular_Last  : Natural;
  package SIO is new Text_IO.Enumeration_IO(Enum => Spectrum);
  package MIO is new Text_IO.Modular_IO    (Num  => Unsigned);
  package IIO is new Text_IO.Integer_IO    (Num  => Integer);
  package FIO is new Text_IO.Float_IO      (Num  => Float);
begin
  Text_IO.Put_Line("The String Values are: ");
  Text_IO.Put("Orange for Enumerated Type          ");
  Text_IO.Put_Line("451  for Integer Type  ");
  Text_IO.Put("360.0  for Float Type               ");
  Text_IO.Put_Line("42  for Unsigned Type  ");
  Text_IO.New_Line;
  -- Example 1; using the Value attribute
  Text_IO.New_Line;
  Text_IO.Put_Line("   >>>> Example 1; Using 'Value Attribute <<<< ");
  Color := Spectrum'Value(Text_Color);
  Num   := Integer'Value(Text_Integer);
  FNum  := Float'Value(Text_Float);
  MNum  := Unsigned'Value(Text_Unsigned);
  SIO.Put(Color);   Text_IO.New_Line;
  IIO.Put(Num);     Text_IO.New_Line;
  FIO.Put(Fnum);    Text_IO.New_Line;
  MIO.Put(MNum);    Text_IO.New_Line;
  Text_IO.New_Line;
  -- Example 2; using the procedures of pre-instantiated packages
  Text_IO.Put_Line("   >>>> Example 2; using pre-instantiated packages <<<< " );
  Integer_Text_IO.Get(From => Text_Integer,
                      Item => Num,
                      Last => Integer_Last);
  Float_Text_IO.Get(From => Text_Float,
                    Item => FNum,
                    Last => Float_Last);
  Integer_Text_IO.Put(Num); Text_IO.New_Line;
  Float_Text_IO.Put  (FNum, Fore => 3, Aft => 3, Exp => 0);
                      Text_IO.New_Line(2);
  -- Example 3; using your own instantiated packages
```

```
   Text_IO.Put_Line("   >>>> Example 3; Using own instantiations <<<< ");
   Text_IO.New_Line;
   SIO.Get(From => Text_Color, Item => Color, Last => Spectrum_Last);
   MIO.Get(From => Text_Unsigned, Item => MNum, Last => Modular_Last);
   IIO.Get(From => Text_Integer, Item => Num, Last => Integer_Last);
   FIO.Get(From => Text_Float, Item => FNum, Last => Float_Last);
   -- Now Write the Results to the Screen
   SIO.Put(Item => Color);  Text_IO.New_Line;
   IIO.Put(Item => Num);    Text_IO.New_Line;
   FIO.Put(Item => FNum, Fore => 3, Aft => 3, Exp => 0);
   Text_IO.New_Line;
   MIO.Put(Item => MNum);
   Text_IO.New_Line(2);
   Text_IO.Put_Line(" **** End of String_To_Scalar_Demonstration **** ");
 end String_To_Scalar_Demonstration;
```

10.2.2 Converting a Scalar to a String

This program is exactly the opposite of the previous one..

```
with Ada.Text_IO, Ada.Integer_Text_IO, Ada.Float_Text_IO;             --  1
use  Ada;                                                              --  2 May safely use Ada
procedure Scalar_To_String_Demonstration is                            --  3 Convert a string to a scalar object
  type Spectrum is (Red, Orange, Yellow, Green, Blue, Indigo, Violet); --  4 Enumerated type
  type Unsigned is mod 2**8;                                           --  5 Unsigned modular type
  Num   : Integer  := 451;                                             --  6 Combustion point of paper in farenheit
  FNum  : Float    := 360.0;                                           --  7 Don't go off on a tangent
  Color : Spectrum := Blue;                                            --  8 Hmmmm.  "You don't look bluish."
  MNum  : Unsigned := 42;                                              --  9 Life, the Universe, and Everything
  Text  : String(1..10);                                               -- 10
  package SIO is new Text_IO.Enumeration_IO(Enum => Spectrum);         -- 11 Instantiate IO for enumerated type
  package MIO is new Text_IO.Modular_IO    (Num  => Unsigned);         -- 12 Instantiate IO for modular type
  package IIO is new Text_IO.Integer_IO     (Num  => Integer);         -- 13 Instantiate IO for predefined Integer
  package FIO is new Text_IO.Float_IO       (Num  => Float);           -- 14 Instantiate IO for predefined Float
begin                                                                  -- 15
  Text_IO.Put_Line(" Example 1; Using 'Image Attribute ");             -- 17   -- Example 1; using the image attribute
  Text_IO.Put_Line(Spectrum'Image(Color));                             -- 18
  Text_IO.Put_Line(Unsigned'Image(MNum));                              -- 19
  Text_IO.Put_Line(Integer'Image(Num));                                -- 20
  Text_IO.Put_Line(Float'Image(FNum));                                 -- 21
  Text_IO.New_Line;                                                    -- 22
  Text_IO.Put_Line(" Example 2; using pre-instantiated packages " );   -- 24   -- Example 2; pre-instantiated packages
  Integer_Text_IO.Put(Num); Text_IO.New_Line;                          -- 25
  Float_Text_IO.Put  (FNum, Fore => 3, Aft => 3, Exp => 0);            -- 26
  Text_IO.New_Line(2);                                                 -- 27   -- Example 3; own instantiated packages
  Text_IO.Put_Line(" Example 3; Using own instantiations ");           -- 29
  SIO.Put(Color);   Text_IO.New_Line;                                  -- 30
  MIO.Put(MNum);    Text_IO.New_Line;                                  -- 31
  IIO.Put(Num);     Text_IO.New_Line;                                  -- 32
  FIO.Put(FNum, Fore => 3, Aft => 3, Exp => 0);                        -- 33
  Text_IO.New_Line(2);                                                 -- 34
       -- Example 4; convert to text and then print                    -- 35
  Text_IO.Put_Line("Example 4; Convert to text, then print ");         -- 36
  SIO.Put(To => Text, Item => Color);                                  -- 37
 Text_IO.Put_Line(Text);                                               -- 38
  MIO.Put(To => Text, Item => MNum);                                   -- 39
  Text_IO.Put_Line(Text);                                              -- 40
  IIO.Put(To => Text, Item => Num);                                    -- 41
  Text_IO.Put_Line(Text);                                              -- 42
  FIO.Put(To => Text, Item => FNum, Aft => 3, Exp => 0);               -- 43
  Text_IO.Put_Line(Text);                                              -- 44
  Text_IO.New_Line;                                                    -- 45
  Text_IO.Put_Line("End of Image_Demonstration ");                     -- 46
end Scalar_To_String_Demonstration;                                    -- 47
```

Output using the 'Image attributes from Annex K. Leading space for positive values. Leading sign for negative values.

Convert each value to a String and then print it. This is built-in to Ada.Text_IO. Don't write your own version of this

11. Exception Management

Ada was one of the first languages to include exception management as a language feature. Nearly all contemporary languages now have this feature.

Ada has certain predefined exceptions and allows the programmer to declare exceptions specific to the problem being solved. Predefined exceptions from package Standard (Annex A.1) are:

Constraint_Error, Storage_Error, Program_Error, Tasking_Error

Predefined input/output errors in package IO_Exceptions are,

Status_Error, Mode_Error, Name_Error, Use_Error, Device_Error,
End_Error, Data_Error, Layout_Error

Other Annex packages define other kinds of exceptions. You will also find exceptions declared in library packages from various software repositories.

11.1 Handling an Exception (ALRM 11.4)

An exception handler must appear in a **begin**...**end** sequence. Therfore you could have something such as,

Ada comb

```
function  Ohm (Volt, Amp : Float) return Float  is    --  1 Parameterized function declaration;  6.3
   Result : Float := 0.0;                             --  2 Initialized local variable
begin                                                 --  3 Begins sequence of algorithmic statements;       6.3
   Result := Volt / Amp;                              --  4 Simple division operation; cannot divide by zero
exception                                             --  5 If we try to divide by zero, land here.
   when Constraint_Error =>                           --  6 This error is raised on divide-by-zero; handle it here.
        Text_IO.Put_Line("Divide by Zero");           --  7 Display the error  on the console
        raise;                                        --  8 Re-raises the exception after handling it.
end Ohm;                                              --  9 Scope terminator with name of unit   6.3
```

Reminder:
Every Ada program body can be viewed in terms of the Ada comb even if one tooth of the comb is not present.

We do not want to return an invalid value from a function so it is better to raise an exception. Sometimes you want a begin ... exception ... end sequence in-line in other code. To call the function Ohm from a procedure, we would want another exception handler. Since the handler reraised the exception, we need another handler in the calling subprogram.

```
with Ada.Exceptions;                                      --  1 Chapter 11.4.1 ALRM; also, see the end of this chapter
use  Ada;                                                 --  2 OK for use clause on package Ada
procedure  Electric (Amp, Volt : in Float;                --  3 In parameters
                     Resistance : out Float)  is          --  4  Out parameter;  6.3
   function MSG (X :.Exceptions.Exception_Occurrence)     --  5 Profile for Exception_Message function
        return String                                     --  6 Return type for Exception_Message
        renames Exceptions.Exception_Message;             --  7 Rename it to three character  function name
begin                                                     --  8 Begins sequence of algorithmic statements;        6.3
   Resistance := Ohm(Amp => Amp, Volt => Volt);           --  9 Simple division operation; cannot divide by zero
exception                                                 -- 10 If we try to divide by zero, land here.
   when Electric_Error:                                   -- 11 Ada.Exceptions.Exception_Occurrence
   Constraint_Error =>                                    -- 12 This error is raised on divide-by-zero; handle it here.
   Text_IO.Put_Line(MSG(Electric_Error));                 -- 13 See lines 5-7;  renamed Exception_Message function
   Exceptions.Reraise_Occurrence(Electric_Error);         -- 14 Procedure for re-raising the exception by occurrence name
end Electric;                                             -- 15 Scope terminator with name of unit      6.3
```

11.2 Declaring your Own Exceptions

You may also define and raise your own exceptions.

```
with Ada.Exceptions; use Ada;                              --  1 Chapter 11.4.1 ALRM
package  Exception_Manager  is                             --  2 A typical exception/error management package
   Overflow : exception;                                   --  3 Own named exception;  User-defined exception
   Underflow : exception;                                  --  4 Ada exception is not a first class object
   Divide_By_Zero : exception;                             --  5 This could be handy for some applications
   type Exception_Store is tagged limited private;         --  6 A place to store exception occurrences
   type Reference is access all Exception_Store'Class;     --  7 In case you need to reference this in another way
   procedure Save ...                                      --  8 Saves an exception to Exception_Store
   procedure Log ...                                       --  9 Logs an exception
   procedure Display ...                                   -- 10 Displays and exception
private                                                    -- 11 Useful to have more operations before this
   type Exception_Set  is array (1..100)                   -- 12 Array of access values to Exception_Occurrence
            of  Exceptions.Exception_Occurrence_Access;    -- 13 Exception_Occurrence_Access is an access type
   type Exception_Store is tagged                          -- 14 A record containing an array of exceptions
        record                                             -- 15
            Current_Exception : Natural := 0;              -- 16 And index over the Exception_Set
            Exception_Set;                                 -- 17 Instance of type from Lines 12-13
        end record;                                        -- 18
end Exception_Manager;                                     -- 19 Package scope terminator
```

```
with Exception_Manager;                                        --  1 Put Exception_Manager package in scope
package Application is                                         --  2
    type Application_Type is private;                          --  3 Private here is partial definition of type
    procedure Start      (Data : in out Application_Type);     --  4 Create and initialize the application
    procedure Restart  (Data : in out Application_Type);       --  5 If there is an exception, you may need to restart
    procedure Stop       (Data : in out Application_Type);     --  6 Stop the application; may be able to restart
    procedure Cleanup (Data : in out Application_Type);        --  7 When there is an error, call this procedure
    procedure Finalization (Data : in out Application_Type);   --  8 Not be confused with Ada.Finalization
    Application_Exception : exception;                         --  9 Your locally defined exception for this package
private                                                        -- 10  Nothing is public from here forward
    type Application_Type is ... --full definition of type     -- 11 Full definition of the private type
end Application;                                               -- 12 End of the specification unit. Needs a body.
```

In the Application package, any one of the subprograms defined might raise an Application_Exception or some other kind of exception. Since we have not used any of the resources of Exception_Manager, it would be better to defer its context clause (put it in scope) in the package body.

```
with Exception_Manager;                              --  1  Localize the context clause
package  body Application is                         --  2
        -- Implementation code for the package body  --  3
end Application;                                     --  4
```

11.3 Raising Exceptions

There is always the question of whether to raise an exception or not. Exceptions are supposed to be indications that something strange has occurred that cannot be handled with the usual coding conventions. Ada 95 even includes an attribute, X'Valid, to help the developer avoid exceptions on scalar types. Consider this program that uses X'Valid.

First an exception should be visible for the user.

Suppose we have the following visible declaration:
`Compound_Data_Error : exception;`

```
procedure Test_The_Valid_Attribute is          --  1
    type Real is digits 7;                     --  2
    type Number is range 0..32_767;            --  3
    type Compound is                           --  4
        record                                 --  5
          Weight : Real := 42.0;               --  6
```

Scalar types declared within the record definition. X'Valid will not work on a record but can be used on scalar components.

```
              Height : Number;                                  --  7
              Width  : Number;                                  --  8
           end record;                                          --  9
           Data : Compound := (80.0, 64, 97);                   -- 10 Record initilialized with aggregate
    begin                                                       -- 11
       if Data.Weight'Valid then                                -- 12 Test the Weight to see if it is valid
              null;                                             -- 13 Usually some sequence of statements
       elsif    Data.Height'Valid then                          -- 14 Test the Height to see if it is valid
           null;                                                -- 15 Usually some sequence of statements
       elsif    Data.Width'Valid then                           -- 16 Test the Widht to see if it is valid
           null;                                                -- 17 Usually some sequence of statements
       else                                                     -- 18 An else part is usually a good idea
              raise Compound_Data_Error;                        -- 19 Failed all around; raise an exception
       end if;                                                  -- 20
  end Test_The_Valid_Attribute;                                 -- 21
```

Not all Ada designers will agree with the above example. It is your responsibility to decide whether this is an appropriate choice in designing your software. The important consideration is that you may define and raise your own exceptions when you feel it is necessary.

11.4 Package Ada.Exceptions

If you are going to manage your own exceptions, consider using the language-defined package,

```
package Ada.Exceptions is     -- This is an Ada language defined package               --  1 ALRM 11.4.1
   type Exception_Id is private;                                                         --  2
   Null_Id : constant Exception_Id;                                                      --  3
   function Exception_Name(Id : Exception_Id) return String;                             --  4
   type Exception_Occurrence is limited private;                                         --  5
   type Exception_Occurrence_Access is access all Exception_Occurrence;                  --  6
   Null_Occurrence : constant Exception_Occurrence;                                      --  7
   procedure Raise_Exception(E : in Exception_Id; Message : in String := "");            --  8
   function Exception_Message(X : Exception_Occurrence) return String;                   --  9
   procedure Reraise_Occurrence(X : in Exception_Occurrence);                            -- 10

   function Exception_Identity(X : Exception_Occurrence) return Exception_Id;            -- 11
   function Exception_Name(X : Exception_Occurrence) return String;                      -- 12
       -- Same as Exception_Name(Exception_Identity(X)).                                 -- 13
   function Exception_Information(X : Exception_Occurrence) return String;               -- 14
   procedure Save_Occurrence(Target : out Exception_Occurrence;                          -- 15
                      Source : in Exception_Occurrence);                                 -- 16
   function Save_Occurrence(Source : Exception_Occurrence)                               -- 17
                      return Exception_Occurrence_Access;                                -- 18
private                                                                                  -- 19
   ... -- not specified by the language                                                  -- 20
end Ada.Exceptions;                                                                      -- 21
```

12. Generic Components

12.1 Generic Subprograms

Whenever you design an algorithm which can be used for many different types, it is worthwhile to put it in the library as a generic routine. Be sure to let the others on your project know about its existence. Also, there are huge libraries of such algorithms already in place such as the Public Ada Library, PAL, a *labor of love* by Richard Conn, Professor of Computing Science at Monmouth College in New Jersey. Here are a couple of really simple generic subprograms. The next example is a generalization of the Next function shown earlier. First we must define the generic specification.

```
generic                                               -- 1 Reserved word for defining templates
   type Item  is (<>);  -- Any discrete type          -- 2 Generic formal Parameter   (GFP)
function Next (Value :  Item ) return Item;           -- 3 Specification for generic subprogram
```

We would not be allowed to code a generic specification with an is such as,

```
generic                                               -- 1 As in line 1, above
   type Item  is (<>);                                -- 2 As in line 2, above
function Next (Value :  Item ) return Item is         -- 3 Illegal; Specification required
      ...                                             -- 4 body of function
end Next;                                             -- 5 before implementation
```

because any generic subprogram must be first specified as a specification. The specification may actually be compiled or may be declared in the specification of a package.

Then we code the actual algorithm. Notice that the algorithm does not change at all for the earlier version of function Next, even though we may now use it for any discrete data type.

```
function Next (Value : Item) return  Item  is         -- 1 Item is a generic formal parameter
begin                                                 -- 2 No local declarations for this function
   if Item'Succ(Value) = Item'Last then               -- 3 A good use of attribute; see ALRM K/104
      return Item'First;                              -- 4 ALRM 6.3
   else                                               -- 5 ALRM 5.3
      return Item'Succ(Value);                        -- 6 Note two returns;  may not be good idea
   end if;                                            -- 7 ALRM 5.3
end Next;                                             -- 8 Always include the function identifier
```

This can be instantiated for any data type. Given the following types, write a few little procedures to cycle through the types,

```
type Month is (January, Februrary, March, April, May, June, July, August, September, October, November, December);
type Color is (Red, Orange, Yellow, Green, Blue, Indigo, Violet) ;  -- our friend, Roy G.  Biv.
type Day is (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday);
type Priority is (Very_Low, Low, Sorta_Medium, Medium, Getting_Higher,  High, Very_High, The_Very_Top);
```

The next generic subprogram is also quite simple. Here we have the famous Swap procedure. Recall that any private type has the predefined operations, =, /=, and assignment. Also, nearly every other Ada data type also has those predefined. The only types without these operations are limited types such as limited private, limited records, tasks, and protected types. Therefore, we can instantiate the Swap procedure with nearly any type in Ada.

```
generic                                               -- 1
   type Element (<>)  is private;                     -- 2 Unconstrained generic parameter
procedure Swap (Left, Right :  in  out Element) ;     -- 3
```

Then we code the actual algorithm. Notice that the algorithm does not change at all even though we may now use it for any non-limited data type.

```
procedure Swap (Left, Right :  in  out Element)  is           -- 1
     Temp : Element := Left;                                   -- 2 Must be constrained in declaration
begin                                                          -- 3
     Left := Right;                                            -- 4
     Right := Temp;                                            -- 5
end Swap;                                                      -- 6
```

An algorithm does not get much easier than the Swap procedure just shown. However, it should be clear from seeing it that you can use this technique to generalize hundreds of other algorithms on your own projects. You can also use this idea to share code with your colleagues.

When you have a lot of generic subprograms for your application, it is often useful to collect those with some common properties into an Ada package. For example, using those already described,

```
package Utilities is
     generic
          type Item is private;                    -- A constrained generic formal parameter
     procedure Swap(L, R : in out Item);

     generic
          type Item is (<>);                       -- A discrete type generic formal parameter
     function Next (Data : Item) return Item;

     generic
          type Item is (<>);                       -- A discrete type generic formal parameter
     function Prev (Data : Item) return Item;

     -- more generic subprograms as appropriate

end Utilities;
```

The Utilities package can be used to collect common algorithms, thereby making up a set of reusable components that can be used to create even larger components. Build generics from other generics.

12.2 Other Generic Formal Parameters

A generic formal type parameter is possible for any type. This includes access types, derived types, array types, and even limited types. For limited types, the designer must include a corresponding set of generic formal operations. Even for other types, generic formal operations are often useful. Consider this private type.

```
generic
     type Item is private;
     with function ">" (L, R : Item ) return Boolean;
     with function "<" (L, R : Item) return Boolean;
package Doubly_Linked_Ring_1  is
     -- Specification of a Doubly_Linked_Ring data strructure
end Doubly_Linked_Ring_1;
```

In the example for the Doubly_Linked_Ring_1, we know that implementation requires some operations beyond simple test for equality. The only operator predefined for a private type is test for equality. Consequently, we may include parameters for other operators. These are instantiated by the client of the package. Before showing the instantiation of this example, we provide the following example that is preferred by many designers of resuable generic data structure components.

```
generic
   type Item is private;
   type Item_Reference is access all Item;
   with function Is_Equal (L, R : Item ) return Boolean;
   with function Is_Less_Than (L, R : Item) return Boolean;
   with function Is_Greater_Than (L, R : Item) return Boolean;
package Doubly_Linked_Ring_2  is
   type Ring is limited private;
   -- Specification of a Doubly_Linked_Ring data strructure
end Doubly_Linked_Ring_2;
```

Even though test for equality is predefined for a private type, the test is on the binary value of the data not on its selected components. If the actual parameter is a record or constrained array, a pure binary comparison may not give the intended result. Instead, by supplying a generic formal parameter, the client of the generic package can ensure the structure is organized according to a given record key. Also, by including an access type for the generic formal private type, the client may have lists of lists, trees of queues, lists of rings, etc. The following example instantiates the Doubly_Linked_Ring_2.

```
with Doubly_Linked_Ring_2 ;
procedure Test_Doubly_Linked_Ring_2 is
   type Stock is record
      Stock_Key :  Positive;
      Description : String (1..20);
   end record;
   type Stock_Reference is access all Stock;
   function Is_Equal (L, R : Stock) return Boolean is
   begin
      return L.Key = R.Key;
   end Is_Equal;
   function ">" ...  -- Overload ">"  Implement using the model of Is_Equal
   function "<" ...
   package Stockkeeper  is new Doubly_Linked_Ring_2( Item => Stock,
                                                   Item_Reference => Stock_Reference,
                                                   Is_Equal => Is_Equal,
                                                   Is_Less_Than => "<",
                                                   Is_Greater_Than => ">");
   The Ring : Stockkeeper.Ring;
   The_Data  : Stock;
begin
   -- Insert and remove stuff from the Ring
end Test_Doubly_Linked_Ring_2;
```

Sometimes it is convenient to combine a set of generic formal parameters into a signature package. A signature package can be reused over and over to instantiate many different kinds of other generic packages. A signature package will often have nothing in it except the generic parameters. It must be instantiated before it can be used. This is an advanced topic. Here is one small, oversimplified, example, derived and expanded from the Ada 95 Language Rationale.

```
package Mapping_Example is -- Begin the enclosing package specification          -- 1
 generic                                                                          -- 2
  type Mapping_Type is private;                                                   -- 3
  type Key is limited private;                                                    -- 4
  type Value is limited private;                                                  -- 5
  with procedure Add    (M : in out Mapping_Type; K : in Key; V : in Value);      -- 6
  with procedure Remove (M : in out Mapping_Type; K : in Key; V : in Value);      -- 7
  with procedure Apply  (M : in out Mapping_Type; K : in Key; V : in Value);      -- 8
package Mapping is end Mapping;                                                   -- 9
   -- Now declare the specification for the generic procedure in the same package
```

Note the generic formal parameters for the signature package, Mapping. The package contains no other operations. This is legal and handy

```
     generic                                                                          -- 10
        with package Mapping_Operations is new Mapping (<>);                          -- 11
          -- This is a generic formal package parameter instead of a generic formal subprogram   -- 12
       procedure Do_Something(M : in out Mapping_Type; K : in Key; V : in Value);     -- 13
  end Mapping_Example; -- End of the enclosing package specification                 -- 14
```

Lines 2 through 9 define the ***generic formal signature*** that will become our generic formal pacakage parameter for the Do_Something procedure. It is important to note that this model has no specification and therefore will not have a body. It is typical of a generic formal model to be nothing more than a set of parameters for later instantiation. The code on Line 11 is the syntax for a generic formal package parameter. The parenthetical box (<>) may have the formal parameters associated with actual parameters if any are visible at this point.

The code beginning on Line 13 is a generic procedure declaration. It is the only procedure in the package specification so it does not represent reality. However, making it a simple procedure with its own formal parameters helps to keep this example simple.

The package body for Mapping_Example will simply implement the procedure Do_Something.

```
package body Mapping_Example is                              --  1
 procedure Do_Something(M : in out Mapping_Type;             --  2
                        K : in Key;                          --  3
                        V : in Value) is                     --  4
 begin -- Do_Something                                       --  5
   Mapping_Operations.Add(M, K, V);                          --  6
 end Do_Something;                                           --  7
end Mapping_Example;                                         --  8
```

We comment the begin statement on Line 5 to emphasize that it belongs to Do_Something. The call on Line 6 is to the Add procedure in the generic formal parameter list for Mapping_Operations. We use dot notation here to emphasize that we are referencing the formal parameter name not the “is new” name. Granted, this example is more of a “do nothing” than a “do something” in spite of its precocious name. However, it will serve to illustrate our first example of the mechanism. Now we can instantiate the units in Mapping_Example

```
with Mapping_Example;                                                         --  1
procedure Test_Mapping_Example is                                             --  2
  Map_Key : Integer := 0;                                                     --  3
  Map_Data : Character := 'A';                                                --  4
  Map_Value : Integer := Map_Key;                                             --  5
    procedure Add    (M : in out Character; K : Integer; V : Integer) is      --  6
    begin                                                                     --  7
        null;                                                                 --  8
    end Add;                                                                  --  9
    procedure Remove (M : in out Character; K : Integer; V : Integer) is      -- 10
    begin                                                                     -- 11
        null;                                                                 -- 12
    end Remove;                                                               -- 13
    procedure Apply (M : in out Character; K : Integer; V : Integer) is       -- 14
    begin                                                                     -- 15
        null;                                                                 -- 16
    end Apply;                                                                -- 17
                                                                              -- 18
  package Character_Mapping is new Mapping_Example.Mapping                    -- 19
          (Mapping_Type => Character,                                         -- 20
           Key          => Integer,                                           -- 21
           Value        => Integer,                                           -- 22
           Add          => Add,                                               -- 23
```

```
           Remove       => Remove,
           Apply        => Apply);
   procedure Do_Something_To_Map
        is new Mapping_Example.Do_Something
           (Mapping_Operations => Character_Mapping);
begin
   Do_Something_To_Map(M => Map_Data,
                       K => Map_Key,
                       V => Map_Value);
end Test_Mapping_Example;
```

12.3 Longer Generic Code Example

Just as you can create simple generic subprograms, as shown above, you can also generalize entire packages. This book has some examples of how to do this. Here is an example of a generic container package which corresponds to some of the the generic packages you will see when programming with Ada.

This package is a ***managed*** FIFO Queue_Manager which includes an ***iterator***. A ***managed data structure*** is one which includes some kind of automatic ***garbage collection***. An ***iterator*** is a mechanism by which you may non-destructively visit every node of a data structure. There are two fundamental kinds of iterators, ***active*** and ***passive***. A ***passive iterator*** is somewhat safer than an active iterator. Also, a passive iterator requires less work from the client. We show a package with an ***active iterator***.

```
with Ada.Finalization;
use  Ada;
generic
  type Element is tagged private;
    -- A more robust design would defined Element as a derivation from Ada.Finalization.Controlled
  with function  Is_Valid(Data : Element) return Boolean;
package Queue_Manager_1 is
     type List is limited private;
     type List_Reference is access all List;
     type List_Item is new Element with private;
     type Item_Reference is access all List_Item'Class;
               -- A classwide access type permitting a heterogenuous queue
     procedure Clear (L : in out List);
     procedure Insert_At_Head (L : in out List; I : in     List_Item'Class);
     procedure Insert_At_Head (L : access List; I : access List_Item'Class);
          -- A more complete design would include added options for the Insert operation
     procedure Copy (Source : in List; Target : in out List);
     function Remove_From_Tail (L : access List) return List_Item'Class;
          -- A more complete design would include added options for the Remove operation
     function "=" (L, R : List) return Boolean;
     function  Node_Count (L : access List) return Natural;
     function  Is_Empty (L : access List) return Boolean;
--    ===============  Define the Active Iterator  =====================
     type Iterator is private;

     procedure Initialize_Iterator(This     : in out Iterator;
                         The_List : access List);
     function  Next(This : in Iterator) return Iterator;

     function  Get (This : in Iterator) return List_Item'Class;
     function  Get (This : in Iterator) return Item_Reference;

     function  Is_Done(This : in Iterator) return Boolean;

     Iterator_Error : exception;
private
     use Ada.Finalization;
     type List_Node;
     type Link is access all List_Node;
     type Iterator is new Link;
```

```
   type List_Item is new Element with null record;
   type List_Node is new Controlled with
      record
         Data : Item_Reference;
         Next : Link;
         Prev : Link;
      end record;
   type List is new Limited_Controlled with
      record
         Count : Natural := 0;
         Head    : Link;
         Tail    : Link;
         Current : Link;
      end record;
   procedure Finalize(One_Node : in out List_Node);
   procedure Finalize(The_List : in out List);
end Queue_Manager_1;
```

An active iterator would require the client to write a loop which successively calls the Next function followed by a Get function. An active iterator is not quite as safe as a passive iterator, but it can be used as an effective building block for contructing passive iterators. Since the list is potentially heterogenuous, the Get returns a classwide type. This can be used in conjuction with dispatching operations. Here is an annotated package body for the above specification. This is a long set of source code but it should be useful to the student because of its near completeness. It also serves as a model for creating other data structures. This package body was compiled using the GNAT Ada compiler.

```
with Text_IO;
with Ada.Exceptions;
with Unchecked_Deallocation;
package body Queue_Manager_1 is
   -- This instantiation enables destruction of unreferenced allocated storage
   procedure Free_Node is new Unchecked_Deallocation
             (Object => List_Node,
              Name   => Link);

   -- This instantiation enables destruction of unreferenced Data items
   procedure Free_Item is new Unchecked_Deallocation
             (Object => List_Item'Class,
              Name   => Item_Reference);

   -- We override Ada.Finalizaion for a single node
   procedure Finalize(One_Node : in out List_Node) is
   begin
      Free_Item (One_Node.Data);
      Free_Node (One_Node.Next);
   end Finalize;

   -- When the list goes out of scope, this is called to clean up the storage
   procedure Finalize(The_List : in out List) is
   begin
      -- Use the Iterator to traverse the list and call Free_Item; add this code yourself
      Free_Node (The_List.Current);
      Free_Node (The_List.Tail);
      Free_Node (The_List.Head);
   end Finalize;

   -- The name says what it does. Note the allocation of a temp. Finalization will
   -- occur to ensure there is no left over storage.
   procedure Insert_At_Head (L : in out List;
                             I : in     List_Item'Class) is
        Temp_Item : Item := new List_Item'(I);
        Temp : Link := new List_Node'(Controlled with
                                      Data => Temp_Item,
                                      Next => null,
                                      Prev => null);
   begin
      if Is_Empty(L'Access)
      then
```

```
       L.Head := Temp;
       L.Tail := Temp;
    else
       L.Head.Prev := Temp;
       Temp.Next := L.Head;
       L.Head := Temp;
    end if;
    L.Count := L.Count + 1;
 end Insert_At_Head;

 -- This is implemented in terms of the non-access version.  Simply makes it convenient
 -- to call this with access to object values, general or storage-pool access values.
 procedure Insert_At_Head (L : access List;
                           I : access List_Item'Class) is
 begin
    Insert_At_Head(L => L.all,
                   I => I.all);
 end Insert_At_Head;

 -- We implement this as a function instead of a procedure with in out modes
 -- because this can be used in an expression to constrain a classwide variable
 -- For example,  X : List_Item'Class := Remove(L);
function Remove_From_Tail (L : access List)
                            return List_Item'Class is
     Result : Item := L.Tail.Data;
 begin
      L.Tail := L.Tail.Prev;
      L.Count := L.Count - 1;
      Free_Item(L.Tail.Next.Data);
      Free_Node(L.Tail.Next);
      return Result.all;
 end Remove_From_Tail;

 -- You might want a more robust "=".  For example, it might be better to traverse
 -- each list, node by node, to ensure that each element is the same.
 function "=" (L, R : List) return Boolean is
 begin
     return L.Count = R.Count;
 end "=";

 -- The name says it.  Simply returns how many nodes in this list.
 function  Node_Count (L : access List) return Natural is
 begin
    return L.Count;
 end Node_Count;

 -- This will not be correct unless you keep careful count of the inserted and deleted nodes.
 function  Is_Empty(L : access List) return Boolean is
 begin
     return L.Count = 0;
 end Is_Empty;

 -- We made List a limited private to prevent automatic assignment.  Instead, we design
 -- this "deep copy" procedure to ensure there will be two separate copies of the data
 procedure Copy (Source : in List;
                 Target : in out List) is
      type Item_Ref is access all List_Item'Class;
      Temp : Link := Source.Tail;
      Local_Data : Item_Reference;
 begin
      Clear(Target); -- Be sure the target is initialized before copying.
      loop
         exit when Temp = null;
         Local_Data := new List_Item'(Temp.Data.all);
         declare
           Local_List_Item
                     : List_Item'Class := Local_Data.all;
         begin
            Insert_At_Head(Target, Local_List_Item);
         end;
         Temp := Temp.Prev;
      end loop;
```

```
   end Copy;

   -- This is pretty simple.  It is also an important part of the overall design.
   procedure Clear (L : in out List) is
   begin
      L.Head    := null;
      L.Tail    := null;
      L.Current := null;
      L.Count   := 0;
   end Clear;

   procedure Initialize_Iterator(This     : in out Iterator;
                                 The_List : access List) is
   begin
      This := Iterator(The_List.Head);
   end Initialize_Iterator;

   function Next(This : access Iterator) return Iterator is
   begin
      return Next(This.all);
   end Next;

   function Next (This : Iterator) return Iterator is
   begin
      return Iterator(This.Next);
   end Next;

   function  Get (This : in Iterator)
                        return List_Item'Class is
   begin
      return This.Data.all;
   end Get;

   function  Get (This : in Iterator) return Item_Reference is
   begin
      return This.Data;
   end Get;

   function  Is_Done(This : in Iterator) return Boolean is
   begin
      return This = null;
   end Is_Done;

   function  Is_Done(This : access Iterator)
                            return Boolean is
   begin
      return Is_Done(This.all);
   end Is_Done;
end Queue_Manager_1;
```

Also need to free data storage in this routine

13. New Names from Old Ones

Renaming is sometimes controversial in Ada programming organizations. Some people like it. Others hate it. The important things to understand are:

1. Renaming does not create new data space. It simply provides a convenient new name for an existing entity.

2. Don't rename the same item over and over with new names. You will simply confuse your colleagues, and probably yourself.

3. Use renaming to simply your code. A new name can sometimes make the code easier to read.

13.1 Making a Long Name Shorter

This section demonstrates some useful ideas such as renaming long package names, commenting the begin statement, getting a line of data from a terminal using Get_Line, and catenating two strings. Also, note that a string may be initialized to all spaces using the ***others*** => aggregate notation.

```
with Text_IO, Ada.Integer_Text_IO;                -- 1 Put Text_IO library unit in scope;           A.10.8/21
procedure  Gun_Aydin  is                          -- 2 "Good morning" in Turkish;                   6.1
   package TIO renames Text_IO;                   -- 3 Shorten a long name with renaming;           8.5.3
   package IIO renames Ada.Integer_Text_IO;       -- 4 Shorter name is same as full name to compiler;          8.5.3
   Text_Data  : String (1..80) := (others => ' ');  -- 5 others => ‘ ‘ iniitalizes string to spaces;  4.3.3
   Len : Natural;                                 -- 4 To be used as parameter in Get_Line;         A.10.7
begin    -- Hello_2                               -- 6 Good idea to comment every begin statement;  2.7/2
   TIO.Put(“Enter Data: “);                       -- 7 Put a string prompt with no carriage return; A.10
   TIO.Get_Line(Text_Data, Len);                  -- 8 After cursor, get a line of text with its length; A.10
   IIO.Put (Len);                                 -- 9 Convert number to text and print it;         A.10 and line 4
   TIO.Put_Line(“   “ & Text_Data(1..Len));       -- 10 Put catenated string with carriage return;  4.4.1
end Gun_Aydin;                                    -- 17 end Label same as procedure name;           6.3
```

13.2 Renaming an Operator ALRM 8.5

Sometimes an operator for a type declared in a *with'ed* package is in scope but not visible . In fact, the rules of Ada are that no entity in scope is actually visible to a client until it is explicitly made visible. An operator is one of the symbol-based operations such as "+", "/", or "=". A use clause for a package will always make these visible, but a use clause also makes too many other things visible. You can selectively import the operators you require through renaming.

Renaming makes a specific operator visible without making all other operators visible. In the following procedure, which draws a diamond on the screen, we rename the packages to make their names shorter and rename the “+” and “-” operators for Text_IO.Count to make them explicitly visible.

```
with ada.text_io;                                          --  1 A.10;  context clause.
with ada.integer_text_Io;                                  --  2 A.10.8/21
procedure diamond1 is                                      --  3 Parameterless procedure
   package TIO renames ada.text_io;                        --  4 Rename a library unit;  8.5.3
   package IIO renames ada.integer_text_io;                --  5 Renames;  8.5.3
   function "+" (L, R : TIO.Count) return TIO.Count        --  6 Rename Operator;  8.5.4
                                   renames TIO."+";        --  7 Makes the  operators directly
   function "-" (L, R : TIO.Count) return TIO.Count        --  8       visible for  "+" and "-" to avoid
                                   renames TIO."-";        --  9        the need for a "use" clause.
   Center : constant TIO.Count     := 37;                  -- 10 type-specific constant; named number
   Left_Col, Right_Col : TIO.Count := Center;              -- 11 type-specific variables
   Symbol : constant Character := 'X';                     -- 12 a character type constant
```

```
     Spacing : TIO.Count := 1;                              -- 13 Local variables for counting
     Increment : TIO.Count := 2;                            -- 14 Initialize the variable
begin    -- Diamond1                                        -- 15 Always declare comment at begin
     TIO.Set_Col(Center);                                   -- 16 Set the column on the screen
     TIO.Put(Symbol);                                       -- 17 Put a character
     for I in 1..8 loop                                     -- 18 begin a for loop with constants
          TIO.New_Line(Spacing);                            -- 19 Advance one line at a time
          Left_Col := Left_Col - Increment;                 -- 20 See lines 8 & 9, above
          Right_Col := Right_Col + Increment;               -- 21 Data type and operator visibility
          TIO.Set_Col(Left_Col);                            -- 22
          TIO.Put(Symbol);                                  -- 23
          TIO.Set_Col(Right_Col);                           -- 24
          TIO.Put(Symbol);                                  -- 25
     end loop;                                              -- 26
     for I in 9..15 loop                                    -- 27
          TIO.New_Line(Spacing);                            -- 28
          Left_Col := Left_Col + Increment;                 -- 29 Increment the Left Column by 1
          Right_Col := Right_Col - Increment;               -- 30 Increment the Right Column by 1
          TIO.Set_Col(Left_Col);                            -- 31 Set the column
          TIO.Put(Symbol);                                  -- 32 Print the symbol
          TIO.Set_Col(Right_Col);                           -- 33 Set the column
          TIO.Put(Symbol);                                  -- 34 Print the symbol
     end loop;                                              -- 35 Loop requires an end loop
     TIO.Set_Col(Center);                                   -- 36 Set the column for final character output
     TIO.Put(Symbol);                                       -- 37 The last character for the diamond
end Diamond1;                                               -- 38 End of scope and declarative region
```

You may want to plan ahead for ease of operator usage through careful package design. In the following example, the operators are renamed in a nested package which can be made visible with a use clause.

```
package Nested is                                                 --  1  Package specification
     type T1 is private;  -- this is called a partial view of the type   --  2  Only =, /=, and :=
     type Status is (Off, Low, Medium, High, Ultra_High, Dangerous);    --  3  Enumerated type; full set
     -- operations on T1 and Status                               --  4     of infix operators is available
     package Operators is                                         --  5  A nested package specification
          function “>=“ (L, R : Status) return Boolean            --  6  Profile for a function and
                         renames Nested.”>=“;                      --  7        renames for the >= operator
          function “=“ (L, R : Status) return Boolean             --  8  Profile for an = function and
                         renames Nested.” =“;                      --  9 renames of the = operator
     end Operators;                                               -- 10  Nested specifcation requires end
private                                                           -- 11  Private part of package
     type T1 is ...                                               -- 12  Full definition of type from line 2
end Nested;                                                       -- 13  Always include the identifier
```

The above package can be accessed via a “with Nested;” context clause followed by a “use Nested.Operators;” to make the comparison operators explicitly visible. Not everyone will approve of this approach, but it has been employed in many Ada designs to simplify the use of infix operators because it eliminates the need for localized renaming. We caution you to use this technique with discretion.

```
with Nested;                                          -- 1 Always include the identifier
procedure Test_Nested is                              -- 2 A simple procedure body
     use Nested.Operators;                            -- 3 Use clause for  nested package
     X, Y : Nested.Status := Nested.Status'First;     -- 4 Declare some Status objects
begin  -- Test_Nested                                 -- 5 Always include Identifier
     -- Get some values for X, and Y                  -- 6 This code is commented
     if X = Nested.Status'Last then                   -- 7 = is made visible with line 3
          -- Some statements here                     -- 8 Comments again
     end if;                                          -- 9 Of course.  End if required
end Test_Nested;                                      --10 Always use identifier with end
```

The code just shown illustrates a technique for letting the client make the selected operators visible via a use clause on the nested package specification. This is actually a better solution than the *use type* (ALRM 8.4/4) because it only makes a restricted set of operators visible. The downside of this is that it requires the designer to think ahead. Thinking ahead is probably an unreasonable expectation of designers.

13.3 Renaming an Exception

Sometimes it is useful to rename an exception locally to where it will be used. For example,

```
with Ada.IO_Exceptions;
package My_IO is
   -- various IO services
   -- Data_Error : exception renames Ada.IO_Exceptions.Data_Error;
   ...
end My_IO;
```

13.4 Renaming a Component

One of the most frequently overlooked features of Ada renaming is the option of giving a component of a composite type its own name.

```
with Ada.Text_IO;
package Rename_A_Variable is
   -- various IO services
   -- Record_Count : renames Ada.Text_IO.Count;
   ...
end Rename_A_Variable;
```

13.4.1 Renaming an Array Slice

Suppose you have a string,

```
Name : String(1..60);
```

where 1..30 is the last name, 31..59 is the first name and 60 is the middle initial. You could do the following.

```
declare
   Last     : String  renames Name(1..30);
   First    : String  renames Name(31.29);
   Middle : String  renames Name(60.60);
begin
   Ada.Text_IO.Put_Line(Last);
   Ada.Text_IO.Put_Line(First);
   Ada.Text_IO.Put_Line(Middle);
end;
```

where each Put_Line references a named object instead of a range of indices. Notice that the object still holds the same indices. Also, the renamed range constrains the named object. No new space is declared. The renaming simply gives a new name for existing data.

13.4.2 Renaming a Record Component

Consider the following definitions,

```
subtype Number_Symbol is Character range '0'..'9';
subtype Address_Character is Character range Ada.Characters.Latin_1.Space
                          .. Ada.Characters.Latin_1.LC_Z;
type Address_Data is array(Positive range <>) of Address_Character;
type Number_Data is array(Positive range <>) of Number_Symbol;
type Phone_Number is record
   Country_Code : Number_Data(1..2);
   Area_Code : Number _ Data (1..3);
   Prefix  : Number_ Data (1..3);
   Last_Four : Number_ Data (1..4);
end record;
type Address_Record is
   The_Phone : Phone_Number;
   Street_Address_1 :  Address_Data(1..30);
   Street_Address_2 :  Address_Data(1..20);
   City    : Address_Data (1..25);
   State  : Address_Data(1..2);
   Zip : Number _ Data (1..5);
   Plus_4 : Number_ Data (1..4);
end record;

One_Address_Record : Address_Record;
```

Now you can rename an inner component for direct referencing in your program. For example, to rename the Area_Code in a declare block,

```
declare
   AC : Number_ Data renames One_Address_Record .The_Phone.Area_Code;
begin
   null;
end;
```

The declaration of AC does not create any new data space. Instead, it localizes the name for the component nested more deeply within the record. If the record had deeply nested components that you needed in an algorithm, this renaming could be a powerful technique for simplifying the names within that algorithm.

13.5 Renaming a Library Unit

Suppose you have a package in your library that everyone on the project uses. Further, suppose that package has a long name. You can with that library unit, rename it, and compile it back into the libaray with the new name. Anytime you with the new name, it is the same as withing the original.

```
-- The following code compiles a renamed library unit into the library
with Ada.Generic_Elementary_Functions;
package Elementary_Functions renames Ada.Generic_Elementary_Functions;

with Graphics.Common_Display_Types;
package CDT renames Graphics.Common_Display_Types;
```

Take care when doing this kind of thing. You don't want to confuse others on the project by making up new names that no one knows about. Also, renaming can be a problem when the renamed entity is too far from its origins.

13.6. Renaming an Object or Value

This can be especially troublesome when done too often. I recall a project where the same value was renamed about seven times throughout a succession of packages. Each new name had meaning within the context of the new package but was increasingly untraceable the further one got from its original value.

```
package Messenger is                                           --  1  Specification Declaration
   type Message is tagged private;                             --  2 Partial definition , tagged type
   type Message_Pointer is access all Message'Class;           --  3 Classwide pointer
   procedure Create(M : in out Message;                        --  4 Operation on the type
                    S : in String);                            --  5 Second parameter for Operation
   procedure Clear (M : in out Message);                       --  6 Clear all fields of the Message
   function  Message_Text (M : Message) return String;         --  7 Return the Data of Message
   function  Message_Length(M : Message) return Natural;       --  8 Return the Length of Message
private                                                        --  9 Private part of specification
   type String_Pointer is access all String;                   -- 10 Private pointer declaration
   type Message is tagged record                               -- 11 Full definition of  type Message
      Data   : String_Pointer;                                 -- 12 Component of Message
      Length : Natural;                                        -- 13 Component of Message
   end record;                                                 -- 14 Ends scope of Message
end Messenger;                                                 -- 15 End scope of specification
```

14. Concurrency with Tasking

Ada is unique among general purpose programming languages in its support for concurrency. There are two models for Ada concurrency: multitasking, and distributed objects. The latter, distributed objects is beyond the scope of this book. We focus our discussion on multitasking. In Ada this is simply called tasking. Tasking is implemented using standard Ada language syntax and semantics along with two additional types: task types and protected types. The syntax and semantics of ***task*** types and ***protected*** types is described in Chapter 9 of the Ada Language Reference Manual (ALRM). The semantics are augmented in Annex D and Annex C of the ALRM.

Each task is a sequential entity that may operate concurrently with other tasks. A task object may be either an anonymous type or an object of a task type.

14.1 A Keyboard Entry Example

The following tasks are anonymous types, and will operate concurrently.

```
package Set_Of_Tasks is
   task T1;                              -- object of anonymous task type
   task T2 is                            -- communicating object
      entry A;                           -- entry point to task
      entry B;                           -- entry point to task
   end T2;                               -- end of task specification
   task T3 is                            -- communicating task object
      entry X(I : in Character);         -- parameterized entry point
      entry Y(I : out Character);        -- parameterized entry point
   end T3;                               -- end of task specification
end Set_Of_Tasks;                        -- end of package specification
```

A task has two parts: specification and body. A task may not be a library unit and cannot be compiled by itself. A task must be declared inside some other library unit. In the example, above, there are three task specifications within a package specification. The body of each task will be within the body of the package. For example,

```
with Ada.Text_IO;                            --  1 Context clause
with Ada.Characters.Latin_1;                 --  2 For referencing special characters
use  Ada;                                    --  3 Make package Ada visible
use  Characters;                             --  4 Make package Characters visible
package body Set_Of_Tasks is                 --  5 Enclosing scope for the task bodies
   task body T1 is                           --  6 Implement task T1
     Input  : Character;                     --  7 Local variable
     Output : Character;                     --  8 Local variable
     Column : Positive := 1;                 --  9 Could be Text_IO.Positive_Count
   begin                                     -- 10
     loop                                    -- 11
      Text_IO.Get_Immediate (Input);         -- 12 Input character with no return key entry
      exit when Input = '~';                 -- 13 If the character is a tilde, exit the loop
      T3.X(Input);                           -- 14 Put entry in queue for T3.X; suspend
      T2.A;                                  -- 15 Put entry in queue for T2.A; suspend
      T2.B;                                  -- 16 Put entry in queue for T2B; suspend
      T3.Y(Output);                          -- 17 Put entry in queue for T3.Y; suspend
      if Column > 40 then                    -- 18 No more than 40 characters per line
         Column := 1;                        -- 19 Start the character count over from 1
         Text_IO.New_Line;                   -- 20 and then start a new line
      else                                   -- 21
         Column := Column + 1;               -- 22 Increment the character per line count
      end if;                                -- 23
      Text_IO.Set_Col(Text_IO.Positive_Count(Column));   -- 24  Note type conversion here
```

```
         Ada.Text_IO.Put(Output);              -- 25 Print the character on the screen; echo
       end loop;                               -- 26
     end T1;                                   -- 27 End of task T1 implementation
                                               -- 28
     task body T2 is                           -- 29 Implement body of task T2
     begin                                     -- 30
       loop                                    -- 31
         select                                -- 32 Select this alternative or terminate when done
           accept A;                           -- 33 Rendezvous point; corresponds to entry in
           accept B;                           -- 34 task specification.  These are sequential here.
         or                                    -- 35 The alternative to selecting  accept A;
           terminate;                          -- 36 Taken only when nothing can call this anymore
         end select;                           -- 37
       end loop ;                              -- 38
     end T2;                                   -- 39
                                               -- 40
     task body T3 is                           -- 41 Implement task T3 body
       Temp : Character := Latin_1.Nul;        -- 42 Local variable
     begin                                     -- 43
      loop                                     -- 44 Choose rendezvous altenative
        select                                 -- 45 Another selective accept statement
          accept X (I : in     Character ) do  -- 46 Begins critical region for rendezvous
              Temp := I;                       -- 47 Calling task is suspended until end statement
          end X;                               -- 48 Rendezvous complete.  Caller is not suspended
        or                                     -- 49 or this next altenative
          accept Y (I :    out Character ) do  -- 50 Critical region begins with do statement
             I := Temp;                        -- 51 Caller is suspended at this point
             Temp := Latin_1.Nul;              -- 52 The non-printing nul character
          end Y;                               -- 53 Rendezvous complete at this point
        or                                     -- 54 or the terminate alternative which will only
          terminate;                           -- 55 be taken if no other task can call this one
       end select;                             -- 56 end of scope for the select statement
      end loop;                                -- 57
     end T3;                                   -- 58
  end Set_Of_Tasks;                            -- 59
```

We apologize for the length of this example. It does serve to show a lot of interesting issues related to tasking. You can key it in and it will work. We also suggest you experiment with it by little alterations.

Each task is coded as a loop. Task T1 simply gets a character from the keyboard, sends that character to T3, gets it back from T3, and prints it to the screen. T3 does nothing with the character, but it could have more logic for examining the character to see if it is OK. You could modify this program to behave as a simple data entry application. We recommend you do this as an exercise.

Here is a simple little test program you can use with this package.

```
with Set_Of_Tasks;
procedure Test_Set_Of_Tasks is
begin
  null;
end Test_Set_Of_Tasks;
```

The tasks, in package Set_Of_Tasks, will begin executing as soon as the null statement is executed. It is not necessary to call the tasks.

Some tasks will have one or more ***entry*** specifications. In Ada, an entry is unique because it implies an entry queue. That is, a call to an entry simply places an entry into a queue. An entry call is not a request for immediate action. If there are already other entries in that queue, the request for action will have to wait for the entries ahead of it to be consumed. Entries disappear from the queue in one of several ways. The most common is for them to complete the rendezvous request.

Each task has a begin statement. Two of the tasks, T2 and T3, have local variables. The accept statements in the bodies of T2 and T3 correspond to the entry statements in their specifications. A task body may have more than one accept statement for each entry. When an accept statement includes a ***do*** part, everything up to the end of accept statement is called the ***critical region***. A calling task is suspended until the critical region is finished for its entry into the task queue.

Now we examine the details of the program example. Each task specification in the package specification is an anonymous task. We know this because the word type does not appear in the specification. Task T1 is not callable because it has no entries. Task T2 is callable, but has no parameters in the call. T3 is callable and includes a parameter list in each entry. Any call to an entry is nothing more than placement of a request for action in an entry queue.

The body of the package contains the bodies of the corresponding task specifications. Task body T1 is implemented as a loop. This is not a good model for task design. In fact, it is a bad design. However, it does give us an entry point into understanding. A better design would permit interrupts to occur and be handled as they occur rather than within the confines of a loop. We show an example of this kind in the next example.

Line 14 is an entry call to T3.X. It includes a parameter of type Character. This entry call puts a request for action in the T3.X queue. There are, potentially, other entries already in that queue. The default, in Ada, is that the entries will be consumed in a FIFO order. This default may be overridden by the designer when deemed appropriate. At Line 14, Task T1 is suspended while waiting for the completion of its request for action. Task T1 will resume once that request is completed.

Lines 15 and 16 are *do nothing* entry calls. We include them in this example for educational purposes, not because they add anything to the design or performance. If we were to reverse Lines 15 and 16, this program would deadlock. Each task is a sequential process. The two accept statements in task T2 are sequential. Entry B cannot be processed until Entry A is processed. This is an important feature of Ada, and almost all models for communicating sequential processes that operate concurrently.

On line 32 in task T2 and line 45 of task T3, we show the start of a ***select*** statement. This construct allows the task to take a choice of ***accept*** alternatives, depending on which entry is called. The accept statements in task T3 are not sequential. That is, entry X is not dependent on entry Y and entry Y is not dependent on entry X. The corresponding accept statements may proceed regardless of which is called first.

Lines 36 and 56 have the ***terminate*** alternative within a select statement. This alternative will never be taken unless no other task can call one of the other entries. The Ada run-time will take the terminate path for every task that has reached the state where it cannot be called, cannot call any other task, and has no other tasks currently dependent on it. This is a graceful way to for a task to die. There is no need for a special *shutdown* entry. Terminate should be used for most service tasks.

If you do not understand the mechanisms associated with an entry queue, you will not understand communicating tasks. It is a rule that, when a task puts an entry into the queue of another task, that entry remains in the queue until it is consumed or otherwise is removed from the queue. The task that puts the entry is suspended until the request for action is completed. The calling task may request, as part of the call, that the request remain in the queue for a limited period, after which it is removed from the queue.

Task T3 cannot identify who called which entry. It cannot purge its own queue. It can determine how many entries are in each queue. That is, we could have a statement that gets X'Count or Y'Count within task T3.

Lines 47-48 and 52-53 are the procedural statements within an accept statement. Every statement between the word ***do*** and the corresponding ***end*** is in the *critical region*, mentioned earlier. Statement 47 must occur before statement 48. Task T1, when it makes a call, T3.Input(...), is suspended until the entire critical region is finished. T3.Input will consume an entry from its own queue, process that entry in the critical region, and finish. Once it is finished with the statements in the critical region, task T1 is released from its suspended state and may continue.

In tasks T2 and T3, the loop serves a slightly different purpose than in task T1. Here the loop is more of a semantic construct to prevent the task from doing one set of actions and then terminating. That is, the loop guarantees the task will remain active for as long as it is needed.

14.2 Protecting Shared Data

It has been traditional for a design in which concurrent threads share access to the same resource to use some kind of Semaphore. Semaphores come in many different varieties. The two most common are the counting semaphore and the binary semaphore. The latter is sometimes called a Mutex. A Semaphore is a low-level mechanism that exposes a program to many kinds of potential hazards. Ada uses a different mechanism, the protected object, which allows the programmer to design encapsulated, self-locking objects where the data is secure against multiple concurrent updates.

Protected types are a large topic. Therefore, we show only one simple version in this book. The reader is encouraged to study this in greater depth if they need to develop Ada software using the tasking model. The following example illustrates all of three operators of a protected object. There a lot of reasons why you would not want to design a task-based application in exactly the way this one is designed. There are some inherent inefficiencies in the design but it does illustrate some fundamental ideas you should know.

```
with Ada.Text_IO;                                          --  1
procedure Protected_Variable_Example is                    --  2
  package TIO renames Ada.Text_IO;                         --  3
  task T1;                                                 --  4
  task T2;                                                 --  5
  protected Variable is                                    --  6 Could have been a type definition
     procedure Modify(Data : Character);                   --  7 Object is locked for this operation
     function  Query return Character ;                    --  8 Read-only. May not update data
     entry Display(Data : Character; T : String);          --  9 An entry has a queue
  private                                                  -- 10
     Shared_Data : Character := '0';                       -- 11 All data is declared here
  end Variable;                                            -- 12
   protected body Variable is                              -- 13 No begin end part in protected body
     entry Display(Data : Character; T : String)           -- 14 A queue and a required barrier that
        when Display'Count > 0 is                          -- 15    acts like a pre-condition
      begin                                                -- 16
        TIO.Put(T & " ");                                  -- 17
        TIO.Put(Data);                                     -- 18
        TIO.New_Line;                                      -- 19
     end Display;                                          -- 20
     procedure Modify (Data : Character) is                -- 21
     begin                                                 -- 22
        Shared_Data := Data;                               -- 23
     end Modify;                                           -- 24
     function Query return Character is                    -- 25
     begin                                                 -- 26
        return Shared_Data;                                -- 27
     end Query;                                            -- 28
  end Variable;                                            -- 29
  task body T1 is                                          -- 30
     Local : Character := 'a';                             -- 31
      Output : Character;                                  -- 32
   begin                                                   -- 33
      loop                                                 -- 34
        TIO.Get_Immediate(Local);                          -- 35
        exit when Local not in '0'..'z';                   -- 36
        Variable.Modify(Local);                            -- 37
         Output := Variable.Query;                         -- 38
```

When a procedure is executed, the object is locked for update only. It is performed in mutual exclusion. No other updates can be performed at the same time. Any other calls to modify must wait for it to be the protected object to be unlocked.

The object is locked for read-only. No updates can be performed. A function is not allowed to update the encapsulated data.

It does not matter how many tasks are trying to update the data. Only one can do so at any time. This task, and its corresponding task will update the protected variable in mutual exclusion.

```
         Variable.Display(Output, "T1 ");              -- 39
      end loop;                                        -- 40
   end T1;                                             -- 41
   task body T2 is                                     -- 42
      Local : Character :='a';                         -- 43
      Output : Character;                              -- 44
   begin                                               -- 45
      loop                                             -- 46
         TIO.Get_Immediate(Local);                     -- 47
          exit when Local not in '0'..'z';             -- 48
         Variable.Modify(Local);                       -- 49
         Output := Variable.Query;                     -- 50
         Variable.Display(Output, "T2 ");              -- 51
      end loop;                                        -- 52
   end T2;                                             -- 53
begin                                                  -- 54
   null;                                               -- 55
end Protected_Variable_Example;  -- 56
```

Every operation in a protected object is performed in mutual exclusion. The object is locked for update only during the modification operations. It is locked for read only during query operations. It is impossible for both update and query to occur at the same time. A function is read-only. During function calls, the object is locked for read-only. An entry, as with a task, has a queue. Every entry is controlled by a boolean pre-condition that must be satisfied before it can be entered.

Think of the difference between a semaphore and a protected type in terms of an airplane lavatory. If you were to enter the lavatory and depend on the flight attendendant to set the lock when you enter and remove the lock to let you out, that would be analogous to a semaphore. In a protected type, once you enter the lavatory, you set the lock yourself. Once you are finished with your business in the lavatory, you unlock it yourself, and it is now free for someone else to use. A protected object knows when it is finished with its work and can unlock itself so another client can enter.

A. Annexes, Appendices and Standard Libraries

Reserved Word List

abort
abs
abstract
accept
access
aliased
all
and
array
at

begin
body

case
constant

declare
delay
delta
digits
do

else
elsif
end
entry
exit

for
function

generic
goto

if
in
is

limited
loop

mod

new
not
null

of
or
others
out

package
pragma
private
procedure
protected

raise
range
record
rem
renames
requeue
return
reverse

select
separate
subtype

tagged
task
terminate
then
type

until
use

when
while
with

xor

Every language has reserved words, sometimes called keywords. Notice that, among Ada's 69 reserved words, there are no explicit data types. Instead, pre-defined types are declared in package Standard.

Sometimes people will try to evaluate a language by counting the number of reserved words. This is a silly metric and the intelligent student will select more substantive criteria.

Some Ada reserved words are overloaded with more than one meaning, depending on context. The compiler will not let you make a mistake in the use of a reserved word.

A.1 Package Standard

Standard is always in scope. Every entity is directly visible. Think of it as the root parent of every other package in any Ada program.

```
package Standard is          -- This package is always visible and never needs a with clause or use clause
   pragma Pure(Standard);
   type Boolean is (False, True); -- An enumerated type; and ordered set;  False is less than True
   -- The predefined relational operators for this type are as follows:
   -- function "="   (Left, Right : Boolean) return Boolean;
   -- function "/="  (Left, Right : Boolean) return Boolean;
   -- function "<"   (Left, Right : Boolean) return Boolean;
   -- function "<="  (Left, Right : Boolean) return Boolean;
   -- function ">"   (Left, Right : Boolean) return Boolean;
   -- function ">="  (Left, Right : Boolean) return Boolean;

   -- The predefined logical operators and the predefined logical
   -- negation operator are as follows:
   -- function "and" (Left, Right : Boolean) return Boolean;
   -- function "or"  (Left, Right : Boolean) return Boolean;
   -- function "xor" (Left, Right : Boolean) return Boolean;
   -- function "not" (Right : Boolean) return Boolean;

   -- The integer type root_integer is predefined; The corresponding universal type is universal_integer.
      type Integer is range implementation-defined;
      subtype Natural  is Integer range 0 .. Integer'Last;
      subtype Positive is Integer range 1 .. Integer'Last;
   -- The predefined operators for type Integer are as follows:

   -- function "="  (Left, Right : Integer'Base) return Boolean;
   -- function "/=" (Left, Right : Integer'Base) return Boolean;
   -- function "<"  (Left, Right : Integer'Base) return Boolean;
   -- function "<=" (Left, Right : Integer'Base) return Boolean;
   -- function ">"  (Left, Right : Integer'Base) return Boolean;
   -- function ">=" (Left, Right : Integer'Base) return Boolean;

   -- function "+"   (Right : Integer'Base) return Integer'Base;
   -- function "-"   (Right : Integer'Base) return Integer'Base;
   -- function "abs" (Right : Integer'Base) return Integer'Base;
   -- function "+"   (Left, Right : Integer'Base) return Integer'Base;
   -- function "-"   (Left, Right : Integer'Base) return Integer'Base;
   -- function "*"   (Left, Right : Integer'Base) return Integer'Base;
   -- function "/"   (Left, Right : Integer'Base) return Integer'Base;
```

Package Standard is the implied parent of every other Ada package. It does not need a ***with*** clause or a ***use*** clause. Every element of package Standard is always visible to every part of every Ada program.

This package defines the types, Integer, Boolean, Float, Character, String, Duration. It also defines two subtypes, Natural and Positive.

All numeric types are implementation dependent. Therefore, do not use predefined numeric types in your Ada program designs. Instead, define your own numeric types with problem-based constraints.

Note: Parameter and return types are Integer'Base rather than Integer.

```
-- function "rem" (Left, Right : Integer'Base) return Integer'Base;
-- function "mod" (Left, Right : Integer'Base) return Integer'Base;

-- function "**"  (Left : Integer'Base; Right : Natural) return Integer'Base;

-- The floating point type root_real is predefined; The corresponding universal type is universal_real.
   type Float is digits implementation-defined;
-- The predefined operators for this type are as follows:
-- function "="   (Left, Right : Float) return Boolean;
-- function "/="  (Left, Right : Float) return Boolean;
-- function "<"   (Left, Right : Float) return Boolean;
-- function "<="  (Left, Right : Float) return Boolean;
-- function ">"   (Left, Right : Float) return Boolean;
-- function ">="  (Left, Right : Float) return Boolean;

-- function "+"   (Right : Float) return Float;
-- function "-"   (Right : Float) return Float;
-- function "abs" (Right : Float) return Float;
-- function "+"   (Left, Right : Float) return Float;
-- function "-"   (Left, Right : Float) return Float;
-- function "*"   (Left, Right : Float) return Float;
-- function "/"   (Left, Right : Float) return Float;

-- function "**"  (Left : Float; Right : Integer'Base) return Float;
```

Warning:
Do not use predefined Float from package Standard in your production programs. This type is useful for student programs but is not well-suited to portable software targeted to some actual production application.

```
-- In addition, the following operators are predefined for the root numeric types:
function "*" (Left : root_integer; Right : root_real) return root_real;
function "*" (Left : root_real;    Right : root_integer) return root_real;
function "/" (Left : root_real;    Right : root_integer) return root_real;
-- The type universal_fixed is predefined.
-- The only multiplying operators defined between fixed point types are:

function "*" (Left : universal_fixed; Right : universal_fixed)
         return universal_fixed;
function "/" (Left : universal_fixed; Right : universal_fixed)
         return universal_fixed;
```

Note: Fixed point arithmetic on root types and universal fixed-point types is defined here. See also ALRM 4.5.5/16-20

```
-- The declaration of type Character is based on the standard ISO 8859-1 character set.
-- There are no character literals corresponding to the positions forcontrol characters.
-- They are indicated in italics in this definition. See 3.5.2.

type Character is
```

See also:
package Ada.Characters
package Ada.Characters.Latin_1
package Ada.Characters.Handling

```
(nul, soh, stx, etx, eot, enq, ack, bel,         --   0 (16#00#) ..  7 (16#07#)
 bs,  ht,  lf,  vt,  ff,  cr,  so,  si,          --   8 (16#08#) .. 15 (16#0F#)
 dle, dc1, dc2, dc3, dc4, nak, syn, etb,         --  16 (16#10#) .. 23 (16#17#)
 can, em,  sub, esc, fs,  gs,  rs,  us,          --  24 (16#18#) .. 31 (16#1F#)
' ', '!', '"', '#', '$', '%', '&', ''',          --  32 (16#20#) .. 39 (16#27#)
'(', ')', '*', '+', ',' ,'-', '.', '/',          --  40 (16#28#) .. 47 (16#2F#)
'0', '1', '2', '3', '4', '5', '6', '7',          --  48 (16#30#) .. 55 (16#37#)
'8', '9', ':', ';', '<', '=', '>', '?',          --  56 (16#38#) .. 63 (16#3F#)
'@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',          --  64 (16#40#) .. 71 (16#47#)
'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',          --  72 (16#48#) .. 79 (16#4F#)
'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',          --  80 (16#50#) .. 87 (16#57#)
'X', 'Y', 'Z', '[', '\', ']', '^', '_',          --  88 (16#58#) .. 95 (16#5F#)
'`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',          --  96 (16#60#) .. 103 (16#67#)
'h', 'I', 'j', 'k', 'l', 'm', 'n', 'o',          -- 104 (16#68#) .. 111 (16#6F#)
'p', 'q', 'r', 's', 't', 'u', 'v', 'w',          -- 112 (16#70#) .. 119 (16#77#)
'x', 'y', 'z', '{', '|', '}', '~', del,          -- 120 (16#78#) .. 127 (16#7F#)
reserved_128, reserved_129, bph, nbh,            -- 128 (16#80#) .. 131 (16#83#)
reserved_132, nel, ssa, esa,                     -- 132 (16#84#) .. 135 (16#87#)
hts, htj, vts, pld, plu, ri, ss2, ss3,           -- 136 (16#88#) .. 143 (16#8F#)
dcs, pu1, pu2, sts, cch, mw, spa, epa,           -- 144 (16#90#) .. 151 (16#97#)
sos, reserved_153, sci, csi,                     -- 152 (16#98#) .. 155 (16#9B#)
st, osc, pm, apc,                                -- 156 (16#9C#) .. 159 (16#9F#)
' ', '¡', '¢', '£', '¤', '¥', '¦', '§'           -- 160 (16#A0#) .. 167 (16#A7#)
'¨', '©', 'ª', '«', '¬', '-', '®', '¯'           -- 168 (16#A8#) .. 175 (16#AF#)
'°', '±', '²', '³', '´', 'µ', '¶', '·'           -- 176 (16#B0#) .. 183 (16#B7#)
'¸','¹', 'º', '»', '¼', '½', '¾', '¿'            -- 184 (16#B8#) .. 191 (16#BF#)
```

Characters beyond the normal 7 bit ASCII format now use 8 bits. Also see Wide-Character

```
    'À', 'Á', 'Â', 'Ã', 'Ä', 'Å', 'Æ', 'Ç'            -- 192 (16#C0#) .. 199 (16#C7#)
    'È', 'É', 'Ê', 'Ë', 'Ì', 'Í', 'Î', 'Ï'            -- 200 (16#C8#) .. 207 (16#CF#)
    'Ð', 'Ñ', 'Ò', 'Ó', 'Ô', 'Õ', 'Ö', '×'            -- 208 (16#D0#) .. 215 (16#D7#)
    'Ø', 'Ù', 'Ú', 'Û', 'Ü', 'Ý', 'Þ', 'ß'            -- 216 (16#D8#) .. 223 (16#DF#)
    'à', 'á', 'â', 'ã', 'ä', 'å', 'æ', 'ç'            -- 224 (16#E0#) .. 231 (16#E7#)
    'è', 'é', 'ê', 'ë', 'ì', 'í', 'î', 'ï'            -- 232 (16#E8#) .. 239 (16#EF#)
    'ð', 'ñ', 'ò', 'ó', 'ô', 'õ', 'ö', '÷'            -- 240 (16#F0#) .. 247 (16#F7#)
    'ø', 'ù', 'ú', 'û', 'ü', 'ý', 'þ', 'ÿ'            -- 248 (16#F8#) .. 255 (16#FF#)

-- The predefined operators for the type Character are the same as for any enumeration type.
-- The declaration of type Wide_Character is based on the standard ISO 10646 BMP character set.
-- The first 256 positions have the same contents as type Character.  See 3.5.2.

type Wide_Character is (nul, soh ... FFFE, FFFF);
```

This is equivalent to Unicode. Can be used for internationalization of a language implementation.

```
package ASCII is ... end ASCII;  -- Obsolescent; see J.5

-- Predefined string types:

type String is array(Positive range <>) of Character;
pragma Pack(String);
-- The predefined operators for this type are as follows:
--   function "="  (Left, Right: String) return Boolean;
--   function "/=" (Left, Right: String) return Boolean;
--   function "<"  (Left, Right: String) return Boolean;
--   function "<=" (Left, Right: String) return Boolean;
--   function ">"  (Left, Right: String) return Boolean;
--   function ">=" (Left, Right: String) return Boolean;
```

Strings of with the same constraint can take advantage of these operators.

```
--   function "&" (Left: String;    Right: String)    return String;
--   function "&" (Left: Character; Right: String)    return String;
--   function "&" (Left: String;    Right: Character) return String;
--   function "&" (Left: Character; Right: Character) return String;
   type Wide_String is array(Positive range <>) of Wide_Character;
   pragma Pack(Wide_String);

-- The predefined operators for Wide_String correspond to those for String
```

This operator is used to catenate arrays to arrays, arrays to components, etc. It is defined for any kind of array as well as for predefined type Strring

```
   type Duration is delta implementation-defined range implementation-defined;
-- The predefined operators for the type Duration are the same as forany fixed point type.
```

Used in delay statements in tasking. See data types in package Calendar, ALRM 9.6

```
-- The predefined exceptions:
   Constraint_Error: exception;
   Program_Error   : exception;
   Storage_Error   : exception;
   Tasking_Error   : exception;
```

These exceptions are predefined in this package. A designer may define more exceptions. Note the absence of Numeric_Error, which is now obsolescent in the current standard.

```
end Standard;
```

A.2 The Package Ada

```
package Ada is
   pragma Pure(Ada);
end Ada
```

package Ada is the parent package for many of the library units. It has no type definitions and no operations. It is nothing more than a placeholder packge that provides a common root (common ancestor) for all of its descendants. As you learn more about parent and child packages, you will understand the value for having one package that is a common root.

The expression, pragma Pure (Ada), is a compiler directive. Pragmas are compiler directives. This directive is of little interest to you at this stage of your study. It will be very important when you being developing larger software systems, especially those that require the Distributed Systems Annex (Annex E).

package Numerics

This is the root package for a variety of numerics packages.

```
package Ada.Numerics is
   pragma Pure(Numerics);
   Argument_Error : exception;
   Pi : constant := 3.14159_26535_89793_23846_26433_83279_50288_41971_69399_37511;
   e  : constant := 2.71828_18284_59045_23536_02874_71352_66249_77572_47093_69996;
end Ada.Numerics;
```

A.5.1 Elementary Functions

Elementary functions are defined as a generic package. This means it must be instantiated before it can be used. Note also that trigonometric functions are in radians. Also, the function "**" is an operator that must be made directly visible before it can be used. We recommend renaming it in the scope where it is required. Also, note that the parameters and return type are Float_Type'Base. This reduces any overflow problems associated with intermediate results in extended expressions.

```
generic
   type Float_Type is digits <>;
package Ada.Numerics.Generic_Elementary_Functions is
   pragma Pure(Generic_Elementary_Functions);
   function Sqrt     (X            : Float_Type'Base)        return Float_Type'Base;
   function Log      (X            : Float_Type'Base)        return Float_Type'Base;
   function Log      (X, Base      : Float_Type'Base)        return Float_Type'Base;
   function Exp      (X            : Float_Type'Base)        return Float_Type'Base;
   function "**"     (Left, Right  : Float_Type'Base)        return Float_Type'Base;

        -- Trigonometric functions default in Radians
   function Sin      (X            : Float_Type'Base)        return Float_Type'Base;
   function Sin      (X, Cycle     : Float_Type'Base)        return Float_Type'Base;
   function Cos      (X            : Float_Type'Base)        return Float_Type'Base;
   function Cos      (X, Cycle     : Float_Type'Base)        return Float_Type'Base;
   function Tan      (X            : Float_Type'Base)        return Float_Type'Base;
   function Tan      (X, Cycle     : Float_Type'Base)        return Float_Type'Base;
   function Cot      (X            : Float_Type'Base)        return Float_Type'Base;
   function Cot      (X, Cycle     : Float_Type'Base)        return Float_Type'Base;
   function Arcsin   (X            : Float_Type'Base)        return Float_Type'Base;
   function Arcsin   (X, Cycle     : Float_Type'Base)        return Float_Type'Base;
   function Arccos   (X            : Float_Type'Base)        return Float_Type'Base;
   function Arccos   (X, Cycle     : Float_Type'Base)        return Float_Type'Base;
   function Arctan   (Y            : Float_Type'Base;
                      X            : Float_Type'Base := 1.0) return Float_Type'Base;
   function Arctan   (Y            : Float_Type'Base;
                      X            : Float_Type'Base := 1.0;
                      Cycle        : Float_Type'Base)        return Float_Type'Base;
   function Arccot   (X            : Float_Type'Base;
                      Y            : Float_Type'Base := 1.0) return Float_Type'Base;
   function Arccot   (X            : Float_Type'Base;
                      Y            : Float_Type'Base := 1.0;
                      Cycle        : Float_Type'Base)        return Float_Type'Base;
   function Sinh     (X            : Float_Type'Base)        return Float_Type'Base;
   function Cosh     (X            : Float_Type'Base)        return Float_Type'Base;
   function Tanh     (X            : Float_Type'Base)        return Float_Type'Base;
   function Coth     (X            : Float_Type'Base)        return Float_Type'Base;
   function Arcsinh  (X            : Float_Type'Base)        return Float_Type'Base;
   function Arccosh  (X            : Float_Type'Base)        return Float_Type'Base;
   function Arctanh  (X            : Float_Type'Base)        return Float_Type'Base;
   function Arccoth  (X            : Float_Type'Base)        return Float_Type'Base;

end Ada.Numerics.Generic_Elementary_Functions;
```

Log default base is natural (e). The base may be other than e.

*For the ** function, you may have a visibility problem. You can solve it by renaming it locally after instantiating the package.*

If cycle is not supplied, the default is in radians.

Float_Type'Base permits an unconstrained result that will not raise a constraint error during intermediate operations. This eliminates spurious range constraint violations in complex expressions.

A.10 Ada.Text_IO (Annotated)

Text_IO enables machine-readable data to be formatted as human-readable data and human-readable data to be conveted to machine-readable. For character and string types, no conversion from internal to external format is required. For all other types, transformations should be done with Text_IO; Some operations are overloaded. Overloading is most common when there are two file destinations for an action: a named file or default standard file.

```
with Ada.IO_Exceptions; -- Declared in Annex A  of the Ada Language Reference Manual
package Ada.Text_IO is  -- Converts human-readable text to machine-readable as well as standard input/output
   type File_Type is limited private;                        -- Internal file handle for a program
   type File_Mode is (In_File, Out_File, Append_File);      -- Controls direction of data flow
   type Count is range 0 .. implementation-defined;          -- An integer data type;  see Positive_Count
   subtype Positive_Count is Count range 1 .. Count'Last;    -- May be used freely with type Count
   Unbounded : constant Count := 0;                          -- line and page length
   subtype Field       is Integer range 0 .. implementation-defined;
   subtype Number_Base is Integer range 2 .. 16;             -- Only use: 2, 8, 10 and 16

   type Type_Set is (Lower_Case, Upper_Case);                -- Use this for enumerated types
   -- File Management
   procedure Create  (File : in out File_Type;               -- Program refers to this parameter
                      Mode : in File_Mode := Out_File;       -- Almost always an output file
                      Name : in String    := "";             -- The external name for the file
                      Form : in String    := "");            -- Usage not defined by the language
   procedure Open    (File : in out File_Type;
                      Mode : in File_Mode;       -- May be opened for input or for append
                      Name : in String;
                      Form : in String := "");   -- Rarely used in Ada 95.  Compilers dependent.
```

***Note** overloading of subprogram names from this point on.*

```
   procedure Close   (File : in out File_Type);
   procedure Delete  (File : in out File_Type);
   procedure Reset   (File : in out File_Type; Mode : in File_Mode);   -- Resets the mode of the file
   procedure Reset   (File : in out File_Type);                        -- Resets the mode of the file
   function  Mode    (File : in File_Type) return File_Mode; -- Query the mode of a file
   function  Name    (File : in File_Type) return String;    -- Query the external name of a file
   function  Form    (File : in File_Type) return String;    -- Varies by compiler implementation

   function  Is_Open(File : in File_Type) return Boolean;    -- Query the open status of a file
   -- Control of default input and output files
   procedure Set_Input (File : in File_Type); -- Set this file as the default input file; must be open
   procedure Set_Output(File : in File_Type); -- Set this file as the default ouput file; must be open
   procedure Set_Error (File : in File_Type); -- Use this as the standard error file; must be open
   function Standard_Input  return File_Type; -- Standard input is usually a keyboard
   function Standard_Output return File_Type; -- Standard output is usually a video display terminal
   function Standard_Error  return File_Type;

   function Current_Input   return File_Type; -- Usually the same as Standard Input
   function Current_Output  return File_Type;
   function Current_Error   return File_Type;
   type File_Access is access constant File_Type; -- Enable a pointer value to a file handle
   function Standard_Input  return File_Access;
   function Standard_Output return File_Access;
   function Standard_Error  return File_Access;

   function Current_Input   return File_Access;
   function Current_Output  return File_Access;
   function Current_Error   return File_Access;
```

Access to File_Type has been added to Ada 95 version of Text_IO. This turns out to be quite useful for many situations.

```
   -- Buffer control
   procedure Flush (File : in out File_Type); -- Flushes any internal buffers
   procedure Flush; -- Flush synchronizes internal file with external file by Flushing internal buffers
   -- Specification of line and page lengths
   procedure Set_Line_Length(File : in File_Type; To : in Count);
   procedure Set_Line_Length(To   : in Count);

   procedure Set_Page_Length(File : in File_Type; To : in Count);
   procedure Set_Page_Length(To   : in Count);
   function  Line_Length(File : in File_Type) return Count;
   function  Line_Length return Count;
   function  Page_Length(File : in File_Type) return Count;
   function  Page_Length return Count;
   -- Column, Line, and Page Control
```

Note: *You may use Count instead of Positive_Count but be careful of potential constraint error.*

```
procedure New_Line   (File    : in File_Type;            -- Carriage return/Line Feed for a File
                      Spacing : in Positive_Count := 1); -- Default to 1 unless otherwise called
procedure New_Line   (Spacing : in Positive_Count := 1); -- CR/LF on the default output device
procedure Skip_Line  (File    : in File_Type;            -- Discard characters up to line terminator
                      Spacing : in Positive_Count := 1); -- single line by default
procedure Skip_Line  (Spacing : in Positive_Count := 1);
function  End_Of_Line(File : in File_Type) return Boolean;
function  End_Of_Line return Boolean;

procedure New_Page   (File : in File_Type); -- Terminate current page with page terminator
procedure New_Page;
procedure Skip_Page  (File : in File_Type); -- Discard characters to end of page
procedure Skip_Page;
function  End_Of_Page(File : in File_Type) return Boolean; -- Is this the end of a page?
function  End_Of_Page return Boolean;
function  End_Of_File(File : in File_Type) return Boolean; -- Is this the end of file?
function  End_Of_File return Boolean;

procedure Set_Col (File : in File_Type; To : in Positive_Count); -- Cursor to designated col
procedure Set_Col (To   : in Positive_Count); -- Do not set this to a number less than current Col
procedure Set_Line(File : in File_Type; To : in Positive_Count); -- Cursor to designated line
procedure Set_Line(To   : in Positive_Count); -- Must be value greater than current Line
function Col (File : in File_Type) return Positive_Count; -- What column number in file?
function Col  return Positive_Count;                      -- What column number?
function Line(File : in File_Type) return Positive_Count; -- What line number in file?
function Line return Positive_Count;                      -- What line number?

function Page(File : in File_Type) return Positive_Count; -- What page number in file?
function Page return Positive_Count;                      -- What page number?
-- Character Input-Output
procedure Get(File : in  File_Type; Item : out Character); -- Gets single character from file
procedure Get(Item : out Character);                       -- Gets single character from keyboard
procedure Put(File : in  File_Type; Item : in Character);   -- Put single character; no CR/LF
procedure Put(Item : in  Character);                       -- Put never emits CR/LF

procedure Look_Ahead (File        : in  File_Type; -- Item set to next character without
                      Item        : out Character; -- consuming it.
                      End_Of_Line : out Boolean);  -- False if End of Line/End of Page/End of File
procedure Look_Ahead (Item        : out Character; -- What is next character; don't get it yet
                      End_Of_Line : out Boolean);
procedure Get_Immediate(File      : in  File_Type; -- Get the next character without CR/LF
                        Item      : out Character);
procedure Get_Immediate(Item      : out Character);

procedure Get_Immediate(File      : in  File_Type; -- Only get character if it is available
                        Item      : out Character;
                        Available : out Boolean);  -- False if character is not available
procedure Get_Immediate(Item      : out Character;
                        Available : out Boolean);
-- String Input-Output
procedure Get(File : in  File_Type; Item : out String); -- Get fixed sized string
procedure Get(Item : out String); -- Must enter entire string of size specified

procedure Put(File : in  File_Type; Item : in String); -- Output string;  no CR/LF
procedure Put(Item : in  String);
procedure Get_Line(File : in  File_Type; -- String will vary in size based on value of Last
                   Item : out String;      -- Must be large enough to hold all characters of input
                   Last : out Natural);   -- Number of characters up to line terminator (CR/LF)
procedure Get_Line(Item : out String; Last : out Natural);
procedure Put_Line(File : in  File_Type; Item : in String);
procedure Put_Line(Item : in  String);
```

```
--  Generic packages for Input-Output of any type of signed integer
--  Consider Ada.Integer_Text_IO for standard Integer; you can with that package and get the same result for type Integer.
generic
   type Num is range <>;   --  Parameter for any kind of whole number type except modular type
package Integer_IO is      --  Conversion between human-readable text and internal number format.
   Default_Width : Field := Num'Width;    --  How big is the number going to be?
   Default_Base  : Number_Base := 10;    --  See the options for number base in beginning of Text_IO
   procedure Get(File  : in  File_Type;
                 Item  : out Num;        --  Corresponds to generic formal parameter, above
                 Width : in Field := 0); --  May specify exact number of input characters.
   procedure Get(Item  : out Num;
                 Width : in  Field := 0); --  Should usually leave this as zero

   procedure Put(File  : in File_Type;
                 Item  : in Num;         --  Corresponds to generic formal parameter, above
                 Width : in Field := Default_Width; --  Ordinarily, don't change this
                 Base  : in Number_Base := Default_Base);
   procedure Put(Item  : in Num;
                 Width : in Field := Default_Width;
                 Base  : in Number_Base := Default_Base);
   procedure Get(From : in  String;      --  Get a number from a string value; convert string to integer type
                 Item : out Num;         --  The actual numeric value of the string
                 Last : out Positive);   --  Index value of last character in From
   procedure Put(To   : out String;      --  Get a string from an integer type; convert integer type to string
                 Item : in Num;          --  Can raise a data error, or other IO_Error. Check this first.
                 Base : in Number_Base := Default_Base); --  Consider output in other than base ten.
end Integer_IO;

generic
   type Num is mod <>; --  An unsigned numeric type.  See ALRM 3.5.4/10
package Modular_IO is
   Default_Width : Field := Num'Width;
   Default_Base  : Number_Base := 10;
   procedure Get(File  : in  File_Type;
                 Item  : out Num;
                 Width : in Field := 0);
   procedure Get(Item  : out Num;  Width : in  Field := 0);

   procedure Put(File  : in File_Type;
                 Item  : in Num;
                 Width : in Field := Default_Width;
                 Base  : in Number_Base := Default_Base);
   procedure Put(Item  : in Num;
                 Width : in Field := Default_Width;
                 Base  : in Number_Base := Default_Base);
   procedure Get(From : in  String;
                 Item : out Num;
                 Last : out Positive);
   procedure Put(To   : out String;
                 Item : in Num; --  Get a string from an float type; convert float type to string
                 Base : in Number_Base := Default_Base);
end Modular_IO;
```

Modular_IO is new to Ada 95 and applies to a new Modular data type.

A Modular type is unsigned and has wraparound arithmetic semantics. It is especially useful for array indexes instead of a signed integer type.

```
--  Generic packages for Input-Output of Real Types
generic
   type Num is digits <>;   --  Any floating point type;  ALRM 3.5.7
package Float_IO is
   Default_Fore : Field := 2;              --  Positions to left of decimal point
   Default_Aft  : Field := Num'Digits-1;   --  Positions to right of decimal point
   Default_Exp  : Field := 3;              --  For scientific notation; often zero is OK
   procedure Get(File  : in  File_Type;
                 Item  : out Num;
                 Width : in  Field := 0); --  May specify exact width; usually don't; leave as zero
   procedure Get(Item  : out Num;
                 Width : in  Field := 0);

   procedure Put(File : in File_Type;
                 Item : in Num;
```

```
                 Fore : in Field := Default_Fore;
                 Aft  : in Field := Default_Aft;
                 Exp  : in Field := Default_Exp);
   procedure Put(Item : in Num;
                 Fore : in Field := Default_Fore;
                 Aft  : in Field := Default_Aft;
                 Exp  : in Field := Default_Exp);


   -- Use these procedures to convert a floating-point value to a string or a string to a floating-point value
   procedure Get(From : in String;    -- Get floating point value from a string value
                 Item : out Num;      -- Converts a valid floating point string to a float value
                 Last : out Positive);
   procedure Put(To   : out String;   -- Write a floating point value into an internal string
                 Item : in Num;       -- Converts a  floating point value to a variable of type String
                 Aft  : in Field := Default_Aft;
                 Exp  : in Field := Default_Exp);
end Float_IO;

generic
   type Num is delta <>;   -- Input/Output of fixed point numeric types
package Fixed_IO is

   Default_Fore : Field := Num'Fore;
   Default_Aft  : Field := Num'Aft;
   Default_Exp  : Field := 0;
   procedure Get(File  : in  File_Type;
                 Item  : out Num;
                 Width : in  Field := 0);
   procedure Get(Item  : out Num;
                 Width : in  Field := 0);
   procedure Put(File : in File_Type;
                 Item : in Num;
                 Fore : in Field := Default_Fore;
                 Aft  : in Field := Default_Aft;
                 Exp  : in Field := Default_Exp);
   procedure Put(Item : in Num;
                 Fore : in Field := Default_Fore;
                 Aft  : in Field := Default_Aft;
                 Exp  : in Field := Default_Exp);
   -- Use these procedures to convert a fixed-point value to a string or a string to a fixed-point value
   procedure Get(From : in  String;
                 Item : out Num;
                 Last : out Positive);
   procedure Put(To   : out String;
                 Item : in Num;
                 Aft  : in Field := Default_Aft;
                 Exp  : in Field := Default_Exp);
end Fixed_IO;

generic
   type Num is delta <> digits <>;
package Decimal_IO is -- Decimal types are used for financial computing.

   Default_Fore : Field := Num'Fore;
   Default_Aft  : Field := Num'Aft;
   Default_Exp  : Field := 0;
   procedure Get(File  : in  File_Type;
                 Item  : out Num;
                 Width : in  Field := 0);
   procedure Get(Item  : out Num;
                 Width : in  Field := 0);
   procedure Put(File : in File_Type;
                 Item : in Num;
                 Fore : in Field := Default_Fore;
                 Aft  : in Field := Default_Aft;
                 Exp  : in Field := Default_Exp);
   procedure Put(Item : in Num;
                 Fore : in Field := Default_Fore;
                 Aft  : in Field := Default_Aft;
                 Exp  : in Field := Default_Exp);
```

See: ALRM Annex F
ALRM 3.5.9/4, ALRM 3.5.9/16

A decimal type is a special kind of fixed-point type in which the delta must be a power of ten. This is unlike a normal fixed point type where the granluarity is a power of two.

Decimal types are more accurate for monetary applications and others that can be best served using power of ten decimal fractions.

```
      -- Use these procedures to convert a decimal value to a string or a string to a decimal value
      procedure Get(From : in  String;
                    Item : out Num;
                    Last : out Positive);
      procedure Put(To   : out String;
                    Item : in Num;
                    Aft  : in Field := Default_Aft;
                    Exp  : in Field := Default_Exp);
   end Decimal_IO;

   -- Generic package for Input-Output of Enumeration Types
   generic
      type Enum is (<>);   -- Actual must be a discrete type
   package Enumeration_IO is

      Default_Width   : Field := 0;
      Default_Setting : Type_Set := Upper_Case;
      procedure Get(File : in  File_Type;
                    Item : out Enum);
      procedure Get(Item : out Enum);
      procedure Put(File  : in File_Type;
                    Item  : in Enum;
                    Width : in Field    := Default_Width;
                    Set   : in Type_Set := Default_Setting);
      procedure Put(Item  : in Enum;
                    Width : in Field    := Default_Width;
                    Set   : in Type_Set := Default_Setting);
      -- Use these procedures to convert a enumerated value to a string or a string to a enumerated value
      procedure Get(From : in  String;
                    Item : out Enum;
                    Last : out Positive);
      procedure Put(To   : out String;
                    Item : in  Enum;
                    Set  : in  Type_Set := Default_Setting);
   end Enumeration_IO;
```

An enumerated type is an ordered set of values for a named type. Example:

type Color **is** (Red, Yellow, Blue);
type Month **is** (Jan, Feb,.., Dec)
... is not legal Ada
type Day is (Monday, Tuesday, ...);
type Priority **is** (Low, Medium, High);

```
   -- Exceptions
   Status_Error : exception renames IO_Exceptions.Status_Error;
   Mode_Error   : exception renames IO_Exceptions.Mode_Error;
   Name_Error   : exception renames IO_Exceptions.Name_Error;
   Use_Error    : exception renames IO_Exceptions.Use_Error;
   Device_Error : exception renames IO_Exceptions.Device_Error;
   End_Error    : exception renames IO_Exceptions.End_Error;
   Data_Error   : exception renames IO_Exceptions.Data_Error;
   Layout_Error : exception renames IO_Exceptions.Layout_Error;
```

-- *from package IO_Exceptions*

```
private
   ... -- not specified by the language
end Ada.Text_IO;
```

Ada.Stream_IO

Permits input/ouput of data in terms of System.Storage_Unit.   Use this with attributes: S'Input, S'Output, S'Read, S'Write.   This package makes it possible to store a tag of a tagged type along with the rest of the data in the object.

```
with Ada.IO_Exceptions;
package Ada.Streams.Stream_IO is
   type Stream_Access is access all Root_Stream_Type'Class;
   type File_Type is limited private;
   type File_Mode is (In_File, Out_File, Append_File);
   type Count is range 0 .. implementation-defined;
   subtype Positive_Count is Count range 1 .. Count'Last;
      -- Index into file, in stream elements.
   procedure Create (File : in out File_Type;
                     Mode : in File_Mode := Out_File;
                     Name : in String     := "";
                     Form : in String     := "");
   procedure Open  (File : in out File_Type;
                    Mode : in File_Mode;
                    Name : in String;
                    Form : in String := "");
   procedure Close  (File : in out File_Type);
   procedure Delete (File : in out File_Type);
```

```
   procedure Reset  (File : in out File_Type; Mode : in File_Mode);
   procedure Reset  (File : in out File_Type);
   function Mode (File : in File_Type) return File_Mode;
   function Name (File : in File_Type) return String;
   function Form (File : in File_Type) return String;
   function Is_Open     (File : in File_Type) return Boolean;
   function End_Of_File (File : in File_Type) return Boolean;
   function Stream (File : in File_Type) return Stream_Access;
   -- Return stream access for use with T'Input and T'Output
   -- Read array of stream elements from file
   procedure Read (File : in  File_Type;
                   Item : out Stream_Element_Array;
                   Last : out Stream_Element_Offset;
                   From : in  Positive_Count);
   procedure Read (File : in  File_Type;
                   Item : out Stream_Element_Array;
                   Last : out Stream_Element_Offset);
   -- Write array of stream elements into file
   procedure Write (File : in File_Type;
                    Item : in Stream_Element_Array;
                    To   : in Positive_Count);
   procedure Write (File : in File_Type;
                    Item : in Stream_Element_Array);
   -- Operations on position within file
   procedure Set_Index(File : in File_Type; To : in Positive_Count);

   function Index(File : in File_Type) return Positive_Count;
   function Size (File : in File_Type) return Count;
   procedure Set_Mode(File : in out File_Type; Mode : in File_Mode);
   procedure Flush(File : in out File_Type);
   -- Exceptions
   Status_Error : exception renames IO_Exceptions.Status_Error;
   Mode_Error   : exception renames IO_Exceptions.Mode_Error;
   Name_Error   : exception renames IO_Exceptions.Name_Error;
   Use_Error    : exception renames IO_Exceptions.Use_Error;
   Device_Error : exception renames IO_Exceptions.Device_Error;
   End_Error    : exception renames IO_Exceptions.End_Error;
   Data_Error   : exception renames IO_Exceptions.Data_Error;
private
   ... -- not specified by the language
end Ada.Streams.Stream_IO;
```

Ada.Calendar -- *ALRM 9..6 (also See ALRM, Annex D.8 for Ada.Real-Time calendar package)*

```
  package Ada.Calendar is                                        -- 1
     type Time is private;                                       -- 2
     subtype Year_Number  is Integer range 1901 .. 2099;         -- 3 Ada has always been Y2K compliant
     subtype Month_Number is Integer range 1 .. 12;              -- 4
     subtype Day_Number   is Integer range 1 .. 31;              -- 5
     subtype Day_Duration is Duration range 0.0 .. 86_400.0;     -- 6 Total number of seconds in one day
     function Clock return Time;                                 -- 7
     function Year   (Date : Time) return Year_Number;           -- 8
     function Month  (Date : Time) return Month_Number;          -- 9
     function Day    (Date : Time) return Day_Number;            -- 10
     function Seconds(Date : Time) return Day_Duration;          -- 11

     procedure Split (Date    : in Time;                         -- 12
                      Year    : out Year_Number;                 -- 13
                      Month   : out Month_Number;                -- 14
                      Day     : out Day_Number;                  -- 15
                      Seconds : out Day_Duration);               -- 16
    function Time_Of(Year    : Year_Number;                      -- 17
                     Month   : Month_Number;                     -- 18
                     Day     : Day_Number;                       -- 19
                     Seconds : Day_Duration := 0.0) return Time; -- 20
```

type Duration is defined in package Standard

```
                                                                -- 21
    function "+" (Left : Time;   Right : Duration) return Time; -- 22
    function "+" (Left : Duration; Right : Time) return Time;   -- 23
    function "–" (Left : Time;   Right : Duration) return Time; -- 24
    function "–" (Left : Time;   Right : Time) return Duration; -- 25
    function "<" (Left, Right : Time) return Boolean;           -- 26
    function "<="(Left, Right : Time) return Boolean;           -- 27
    function ">" (Left, Right : Time) return Boolean;           -- 28
    function ">="(Left, Right : Time) return Boolean;           -- 29
    Time_Error : exception;                                     -- 30
private                                                         -- 31
    ... -- not specified by the language                        -- 32
end Ada.Calendar;                                               -- 33
```

System Description Package

Also see: *System.Storage_Elements*
System.Address_To_Access_Conversion
System.Storage_Pools

```
package System is                                                     --  1 Required for every compiler
    pragma Preelaborate(System);                                      --  2 Elaborate at compile time
    type Name is implementation-defined-enumeration-type;             --  3 Look this up for your compiler
    System_Name : constant Name := implementation-defined;            --  4
    -- System-Dependent Named Numbers:                                --  5
    Min_Int                  : constant := root_integer'First;        --  6 root integer is base type
    Max_Int                  : constant := root_integer'Last;         --  7     for all integers in this system
    Max_Binary_Modulus       : constant := implementation-defined;    --  8
    Max_Nonbinary_Modulus    : constant := implementation-defined;    --  9
    Max_Base_Digits          : constant := root_real'Digits;          -- 10
    Max_Digits               : constant := implementation-defined;    -- 11
    Max_Mantissa             : constant := implementation-defined;    -- 12
    Fine_Delta               : constant := implementation-defined;    -- 13
    Tick                     : constant := implementation-defined;    -- 14
    -- Storage-related Declarations:                                  -- 15
    type Address is implementation-defined;                           -- 16 Usually a private type
    Null_Address : constant Address;                                  -- 17
    Storage_Unit : constant := implementation-defined;                -- 18
    Word_Size    : constant := implementation-defined * Storage_Unit; -- 19
    Memory_Size  : constant := implementation-defined;                -- 20
    -- Address Comparison:                                            -- 21
    function "<" (Left, Right : Address) return Boolean;              -- 22
    function "<="(Left, Right : Address) return Boolean;              -- 23
    function ">" (Left, Right : Address) return Boolean;              -- 24
    function ">="(Left, Right : Address) return Boolean;              -- 25
    function "=" (Left, Right : Address) return Boolean;              -- 26
--  function "/=" (Left, Right : Address) return Boolean;             -- 27
    -- "/=" is implicitly defined                                     -- 28
    pragma Convention(Intrinsic, "<");                                -- 29
    ... -- and so on for all language-defined subprograms in this package -- 30
    -- Other System-Dependent Declarations:                           -- 31
    type Bit_Order is (High_Order_First, Low_Order_First);            -- 32 Big-endian/Little-endian
    Default_Bit_Order : constant Bit_Order;                           -- 33
    -- Priority-related declarations (see D.1):                       -- 34
    subtype Any_Priority is Integer range implementation-defined;     -- 35 Used for tasking
    subtype Priority is Any_Priority range Any_Priority'First .. implementation-defined;     -- 36
    subtype Interrupt_Priority is Any_Priority range Priority'Last+1 .. Any_Priority'Last;   -- 37
    Default_Priority : constant Priority := (Priority'First + Priority'Last)/2;             -- 38
private                                                               -- 39
    ... -- not specified by the language                              -- 40
end System;                                                           -- 41
```

Arithmetic operators are defined in package System.Storage_Elements

An implementation may add more specifications and declarations to this package to make it conformant with the underlying system platform.

Annex K (informative): Language-Defined Attributes

Legend for Attribute Prefixes	
P	Subprogram
X	an object
S	type or subtype
E	entry or exception
T	task
R	record
A	array

P’Access For a prefix P that denotes a subprogram:

P’Access yields an access value that designates the subprogram denoted by P. The type of P’Access is an access-to-subprogram type (S), as determined by the expected type. See 3.10.2.

X’Access For a prefix X that denotes an aliased view of an object:

X’Access yields an access value that designates the object denoted by X. The type of X’Access is an access-to-object type, as determined by the expected type. The expected type shall be a general access type. See 3.10.2.

X’Address For a prefix X that denotes an object, program unit, or label:
Denotes the address of the first of the storage elements allocated to X. For a program unit or label, this value refers to the machine code associated with the corresponding body or statement. The value of this attribute is of type System.Address. See 13.3.

S’Adjacent For every subtype S of a floating point type T:
S’Adjacent denotes a function with the following specification:

function S'Adjacent (X, Towards : T) **return** T

If Towards=X, the function yields X; otherwise, it yields the machine number of the type T adjacent to X in the direction of Towards, if that machine number exists. If the result would be outside the base range of S, Constraint_Error is raised. When T’Signed_Zeros is True, a zero result has the sign of X. When Towards is zero, its sign has no bearing on the result. See A.5.3.

S’Aft For every fixed point subtype S:
S’Aft yields the number of decimal digits needed after the decimal point to accommodate the delta of the subtype S, unless the delta of the subtype S is greater than 0.1, in which case the attribute yields the value one. (S’Aft is the smallest positive integer N for which (10**N)*S’Delta is greater than or equal to one.) The value of this attribute is of the type universal_integer. See 3.5.10.

X’Alignment For a prefix X that denotes a subtype or object:
The Address of an object that is allocated under control of the implementation is an integral multiple of the Alignment of the object (that is, the Address modulo the Alignment is zero). The offset of a record component is a multiple of the Alignment of the component. For an object that is not allocated under control of the implementation (that is, one that is imported, that is allocated by a user-defined allocator, whose Address has been specified, or is designated by an access value returned by an instance of Unchecked_Conversion), the implementation may assume that the Address is an integral multiple of its Alignment. The implementation shall not assume a stricter alignment.object is not necessarily aligned on a storage element boundary. See 13.3.

S’Base For every scalar subtype S: S’Base denotes an unconstrained subtype of the type of S. This unconstrained subtype is called the base subtype of the type. See 3.5.

S’Bit_Order For every specific record subtype S: Denotes the bit ordering for the type of S. The value of this attribute is of type System.Bit_Order. See 13.5.3.

P’Body_Version For a prefix P that statically denotes a program unit: Yields a value of the predefined type String that identifies the version of the compilation unit that contains the body (but not any subunits) of the program unit. See E.3.

T’Callable For a prefix T that is of a task type (after any implicit dereference):
Yields the value True when the task denoted by T is callable, and False otherwise; See 9.9.

E’Caller For a prefix E that denotes an entry_declaration:
Yields a value of the type Task_ID that identifies the task whose call is now being serviced. Use of this attribute is allowed only inside an entry_body or accept_statement corresponding to the entry_declaration denoted by E. See C.7.1.

S’Ceiling For every subtype S of a floating point type T:
S’Ceiling denotes a function with the following specification:

```
function S'Ceiling (X : T)  return T
```

The function yields the value éXù, i.e., the smallest (most negative) integral value greater than or equal to X. When X is zero, the result has the sign of X; a zero result otherwise has a negative sign when S'Signed_Zeros is True. See A.5.3.

S'Class For every subtype S of a tagged type T (specific or class-wide):
S'Class denotes a subtype of the class-wide type (called T'Class in this International Standard) for the class rooted at T (or if S already denotes a class-wide subtype, then S'Class is the same as S). S'Class is unconstrained. However, if S is constrained, then the values of S'Class are only those that when converted to the type T belong to S. See 3.9.

S'Class For every subtype S of an untagged private type whose full view is tagged:
Denotes the class-wide subtype corresponding to the full view of S. This attribute is allowed only from the beginning of the private part in which the full view is declared, until the declaration of the full view. After the full view, the Class attribute of the full view can be used. See 7.3.1.

X'Component_Size For a prefix X that denotes an array subtype or array object (after any implicit dereference): Denotes the size in bits of components of the type of X. The value of this attribute is of type universal_integer. See 13.3.

S'Compose For every subtype S of a floating point type T:
S'Compose denotes a function with the following specification:

```
function S'Compose (Fraction : T;
                    Exponent : universal_integer) return T
```

Let v be the value Fraction·T'Machine_Radix**(Exponent–k), where k is the normalized exponent of Fraction. If v is a machine number of the type T, or if ½v½³ T'Model_Small, the function yields v; otherwise, it yields either one of the machine numbers of the type T adjacent to v. Constraint_Error is optionally raised if v is outside the base range of S. A zero result has the sign of Fraction when S'Signed_Zeros is True.

A'Constrained For a prefix A that is of a discriminated type (after any implicit dereference):
Yields the value True if A denotes a **constant**, a value, or a constrained variable, and False otherwise.

S'Copy_Sign For every subtype S of a floating point type T:
S'Copy_Sign denotes a function with the following specification:

```
      function S'Copy_Sign (Value, Sign : T)  return T
```

If the value of Value is nonzero, the function yields a result whose magnitude is that of Value and whose sign is that of Sign; otherwise, it yields the value zero. Constraint_Error is optionally raised if the result is outside the base range of S. A zero result has the sign of Sign when S'Signed_Zeros is True. See A.5.3.

E'Count For a prefix E that denotes an entry of a task or protected unit:
Yields the number of calls presently queued on the entry E of the current instance of the unit. The value of this attribute is of the type universal_integer. See 9.9.

S'Definite For a prefix S that denotes a formal indefinite subtype:
S'Definite yields True if the actual subtype corresponding to S is definite; otherwise it yields False. The value of this attribute is of the predefined type Boolean. See 12.5.1.

S'Delta For every fixed-point subtype S: S'Delta denotes the delta of the fixed-point subtype S.
The value of this attribute is of the type universal_real.

S'Denorm For every subtype S of a floating point type T:
Yields the value True if every value expressible in the form
±mantissa·T'Machine_Radix**(T'Machine_Emin)
where mantissa is a nonzero T'Machine_Mantissa-digit fraction in the number base T'Machine_Radix, the first digit of which is zero, is a machine number (see 3.5.7) of the type T; yields the value False otherwise. The value of this attribute is of the predefined type Boolean. See A.5.3.

S'Digits For every decimal fixed point subtype S:
S'Digits denotes the digits of the decimal fixed point subtype S, which corresponds to the number of decimal digits that are representable in objects of the subtype. The value of this attribute is of the type universal_integer. See 3.5.10.

S'Digits For every floating point subtype S:
S'Digits denotes the requested decimal precision for the subtype S. The value of this attribute is of the type universal_integer. See 3.5.8.

S'Exponent For every subtype S of a floating point type T:
S'Exponent denotes a function with the following specification:

	`function S'Exponent (X : T) return` *universal_integer* The function yields the normalized exponent of X. See A.5.3.
S’External_Tag	For every subtype S of a tagged type T (specific or class-wide): S’External_Tag denotes an external string representation for S’Tag; it is of the predefined type String. External_Tag may be specified for a specific tagged type via an attribute_definition_clause; the expression of such a clause shall be static. The default external tag representation is implementation defined.
A’First(N)	For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype: A’First(N) denotes the lower bound of the N-th index range; its type is the corresponding index type.
A’First	For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype: A’First denotes the lower bound of the first index range; its type is the corresponding index type. See 3.6.2.
S’First	For every scalar subtype S: S’First denotes the lower bound of the range of S. The value of this attribute is of the type of S. See 3.5.
R.C’First_Bit	For a component C of a composite, non-array object R: Denotes the offset, from the start of the first of the storage elements occupied by C, of the first bit occupied by C. This offset is measured in bits. The first bit of a storage element is numbered zero. The value of this attribute is of the type universal_integer. See 13.5.2.
S’Floor	For every subtype S of a floating point type T: S’Floor denotes a function with the following specification: `function S'Floor (X : T) return T` The function yields the value ëXû , i.e., the largest (most positive) integral value less than or equal to X. When X is zero, the result has the sign of X; a zero result otherwise has a positive sign. See A.5.3.
S’Fore	For every fixed point subtype S: S’Fore yields the minimum number of characters needed before the decimal point for the decimal representation of any value of the subtype S, assuming that the representation does not include an exponent, but includes a one-character prefix that is either a minus sign or a space. (This minimum number does not include superfluous zeros or underlines, and is at least 2.) The value of this attribute is of the type universal_integer. See 3.5.10.
S’Fraction	For every subtype S of a floating point type T: S’Fraction denotes a function with the following specification: `function S'Fraction (X : T) return T` The function yields the value X·T’Machine_Radix**(–k), where k is the normalized exponent of X. A zero result, which can only occur when X is zero, has the sign of X. See A.5.3.
E’Identity	For a prefix E that denotes an exception: E’Identity returns the unique identity of the exception. The type of this attribute is Exception_Id. See 11.4.1.
T’Identity	For a prefix T that is of a task type (after any implicit dereference): Yields a value of the type Task_ID that identifies the task denoted by T. See C.7.1.
S’Image	For every scalar subtype S: S’Image denotes a function with the following specification: **function** S'Image(Arg : S'Base) **return** String The function returns an image of the value of Arg as a String. See 3.5.
S’Class’Input	For every subtype S’Class of a class-wide type T’Class: S’Class’Input denotes a function with the following specification: **function** S'Class'Input(Stream : **access** Ada.Streams.Root_Stream_Type'Class) **return** T'Class First reads the external tag from Stream and determines the corresponding internal tag (by calling Tags.Internal_Tag(String’Input(Stream)) — see 3.9) and then dispatches to the subprogram denoted by the Input attribute of the specific type identified by the internal tag; returns that result. See 13.13.2.
S’Input	For every subtype S of a specific type T: S’Input denotes a function with the following specification: **function** S'Input(Stream : **access** Ada.Streams.Root_Stream_Type'Class) **return** T S’Input reads and returns one value from Stream, using any bounds or discriminants written by a corresponding S’Output to determine how much to read. See 13.13.2.

A'Last(N)	For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype: A'Last(N) denotes the upper bound of the N-th index range; its type is the corresponding index type. See 3.6.2.
A'Last	For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype: A'Last denotes the upper bound of the first index range; its type is the corresponding index type. See 3.6.2.
S'Last	For every scalar subtype S: S'Last denotes the upper bound of the range of S. The value of this attribute is of the type of S. See 3.5.
R.C'Last_Bit	For a component C of a composite, non-array object R: Denotes the offset, from the start of the first of the storage elements occupied by C, of the last bit occupied by C. This offset is measured in bits. The value of this attribute is of the type universal_integer. See 13.5.2.
S'Leading_Part	For every subtype S of a floating point type T: S'Leading_Part denotes a function with the following specification: **function** S'Leading_Part (X : T; Radix_Digits : universal_integer) **return** T Let v be the value T'Machine_Radix$^{k-Radix\_Digits}$, where k is the normalized exponent of X. The **function** yields the value $\lfloor X/v \rfloor \cdot v$, when X is nonnegative and Radix_Digits is positive; $\lceil X/v \rceil \cdot v$, when X is negative and Radix_Digits is positive. Constraint_Error is raised when Radix_Digits is zero or negative. A zero result, which can only occur when X is zero, has the sign of X. See A.5.3.
A'Length(N)	For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype: A'Length(N) denotes the number of values of the N-th index range (zero for a null range); its type is universal_integer. See 3.6.2. A'Length For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype: A'Length denotes the number of values of the first index range (zero for a null range); its type is universal_integer. See 3.6.2.
S'Machine	For every subtype S of a floating point type T: S'Machine denotes a function with the following specification: **function** S'Machine (X : T) **return** T If X is a machine number of the type T, the function yields X; otherwise, it yields the value obtained by rounding or truncating X to either one of the adjacent machine numbers of the type T. Constraint_Error is raised if rounding or truncating X to the precision of the machine numbers results in a value outside the base range of S. A zero result has the sign of X when S'Signed_Zeros is True. See A.5.3.
S'Machine_Emax	For every subtype S of a floating point type T: Yields the largest (most positive) value of exponent such that every value expressible in the canonical form (for the type T), having a mantissa of T'Machine_Mantissa digits, is a machine number (see 3.5.7) of the type T. This attribute yields a value of the type universal_integer. See A.5.3.
S'Machine_Emin	For every subtype S of a floating point type T: Yields the smallest (most negative) value of exponent such that every value expressible in the canonical form (for the type T), having a mantissa of T'Machine_Mantissa digits, is a machine number (see 3.5.7) of the type T. This attribute yields a value of the type universal_integer. See A.5.3.
S'Machine_Mantissa	For every subtype S of a floating point type T: Yields the largest value of p such that every value expressible in the canonical form (for the type T), having a p-digit mantissa and an exponent between T'Machine_Emin and T'Machine_Emax, is a machine number (see 3.5.7) of the type T. This attribute yields a value of the type universal_integer. See A.5.3.
S'Machine_Overflows	For every subtype S of a fixed point type T: Yields the value True if overflow and divide-by-zero are detected and reported by raising Constraint_Error for every predefined operation that yields a result of the type T; yields the value False otherwise. The value of this attribute is of the predefined type Boolean. See A.5.4.
S'Machine_Overflows	For every subtype S of a floating point type T:

Yields the value True if overflow and divide-by-zero are detected and reported by raising Constraint_Error for every predefined operation that yields a result of the type T; yields the value False otherwise. The value of this attribute is of the predefined type Boolean. See A.5.3.

S’Machine_Radix For every subtype S of a fixed point type T:
Yields the radix of the hardware representation of the type T. The value of this attribute is of the type universal_integer. See A.5.4.

S’Machine_Radix For every subtype S of a floating point type T:
Yields the radix of the hardware representation of the type T. The value of this attribute is of the type universal_integer. See A.5.3.

S’Machine_Rounds For every subtype S of a fixed point type T:
Yields the value True if rounding is performed on inexact results of every predefined operation that yields a result of the type T; yields the value False otherwise. The value of this attribute is of the predefined type Boolean. See A.5.4.

S’Machine_Rounds For every subtype S of a floating point type T:
Yields the value True if rounding is performed on inexact results of every predefined operation that yields a result of the type T; yields the value False otherwise. The value of this attribute is of the predefined type Boolean. See A.5.3.

S’Max For every scalar subtype S:
S’Max denotes a function with the following specification:
function S'Max(Left, Right : S'Base) **return** S'Base
The function returns the greater of the values of the two parameters. See 3.5.

S’Max_Size_In_Storage_Elements
For every subtype S: Denotes the maximum value for Size_In_Storage_Elements that will be requested via Allocate for an access type whose designated subtype is S. The value of this attribute is of type universal_integer. See 13.11.1.

S’Min For every scalar subtype S:
S’Min denotes a ***function*** with the following specification:
function S'Min(Left, Right : S'Base) **return** S'Base
The function returns the lesser of the values of the two parameters. See 3.5.

S’Model For every subtype S of a floating point type T:
S’Model denotes a function with the following specification:
function S'Model (X : T) **return** T
If the Numerics Annex is not supported, the meaning of this attribute is implementation defined; see G.2.2 for the definition that applies to implementations supporting the Numerics Annex. See A.5.3.

S’Model_Emin For every subtype S of a floating point type T:
If the Numerics Annex is not supported, this attribute yields an implementation defined value that is greater than or equal to the value of T’Machine_Emin. See G.2.2 for further requirements that apply to implementations supporting the Numerics Annex. The value of this attribute is of the type universal_integer.

S’Model_Epsilon For every subtype S of a floating point type T:
Yields the value T’Machine_Radix**(1–T’Model_Mantissa). The value of this attribute is of the type universal_real. See A.5.3.

S’Model_Mantissa For every subtype S of a floating point type T:
If the Numerics Annex is not supported, this attribute yields an implementation defined value that is greater than or equal to éd log(10)/log (T’Machine_Radix)ù+1, where d is the requested decimal precision of T, and less than or equal to the value of T’Machine_Mantissa. See G.2.2 for further requirements that apply to implementations supporting the Numerics Annex. The value of this attribute is of the type universal_integer. See A.5.3.

S’Model_Small For every subtype S of a floating point type T:
Yields the value T’Machine_Radix**(T’Model_Emin–1). The value of this attribute is of the type universal_real. See A.5.3.

S’Modulus For every modular subtype S:
S’Modulus yields the modulus of the type of S, as a value of the type universal_integer. See 3.5.4.

S’Class’Output For every subtype S’Class of a class-wide type T’Class:

S’Class’Output denotes a procedure with the following specification:
procedure S'Class'Output(Stream : **access** Ada.Streams.Root_Stream_Type'Class;
Item : **in** T'Class)
String’Output(Tags.External_Tag(Item’Tag) — see 3.9) and then dispatches to the subprogram denoted by the Output attribute of the specific type identified by the tag. See 13.13.2.

S’Output
For every subtype S of a specific type T:
S’Output denotes a procedure with the following specification:
procedure S'Output(Stream : **access** Ada.Streams.Root_Stream_Type'Class;
Item : **in** T)
S’Output writes the value of Item to Stream, including any bounds or discriminants. See 13.13.2.

D’Partition_ID
For a prefix D that denotes a library-level declaration, excepting a declaration of or within a declared-pure library unit:
Denotes a value of the type universal_integer that identifies the partition in which D was elaborated. If D denotes the declaration of a remote call interface library unit (see E.2.3) the given partition is the one where the body of D was elaborated. See E.1.

S’Pos
For every discrete subtype S:
S’Pos denotes a function with the following specification:
function S'Pos(Arg : S'Base) **return** universal_integer
This function returns the position number of the value of Arg, as a value of type universal_integer. See 3.5.5.

R.C’Position
For a component C of a composite, non-array object R:
Denotes the same value as R.C’Address – R’Address. The value of this attribute is of the type universal_integer. See 13.5.2.

S’Pred
For every scalar subtype S:
S’Pred denotes a function with the following specification:
function S'Pred(Arg : S'Base) **return** S'Base
For an enumeration type, the function returns the value whose position number is one less than that of the value of Arg; Constraint_Error is raised if there is no such value of the type. For an integer type, the function returns the result of subtracting one from the value of Arg. For a fixed point type, the function returns the result of subtracting small from the value of Arg. For a floating point type, the function returns the machine number (as defined in 3.5.7) immediately below the value of Arg; Constraint_Error is raised if there is no such machine number. See 3.5.

A’Range(N)
For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype: A’Range(N) is equivalent to the range A’First(N).. A’Last(N), except that the prefix A is only evaluated once.

A’Range
For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype: A’Range is equivalent to the range A’First.. A’Last, except that the prefix A is only evaluated once. See 3.6.2.

S’Range
For every scalar subtype S:
S’Range is equivalent to the range S’First.. S’Last. See 3.5.

S’Class’Read
For every subtype S’Class of a class-wide type T’Class:
S’Class’Read denotes a procedure with the following specification:
procedure S'Class'Read(Stream : **access** Ada.Streams.Root_Stream_Type'Class; : **out** T'Class)
Dispatches to the subprogram denoted by the Read attribute of the specific type identified by the tag of Item.

S’Read
For every subtype S of a specific type T:
S’Read denotes a procedure with the following specification:
procedure S'Read(Stream : **access** Ada.Streams.Root_Stream_Type'Class;
Item : **out** T)
S’Read reads the value of Item from Stream. See 13.13.2.

S’Remainder
For every subtype S of a floating point type T:
S’Remainder denotes a function with the following specification:
function S'Remainder (X, Y : T) **return** T
For nonzero Y, let v be the value X–n·Y, where n is the integer nearest to the exact value of X/Y; if ½n–X/Y½=½, then n is chosen to be even. If v is a machine number of the type T, the function yields v; otherwise, it yields zero. Constraint_Error is raised if Y is zero. A zero result has the sign of X when S’Signed_Zeros is True. See A.5.3.

S’Round	For every decimal fixed point subtype S: S’Round denotes a function with the following specification: **function** S'Round(X : universal_real) **return** S'Base The function returns the value obtained by rounding X (away from 0, if X is midway between two values of the type of S). See 3.5.10.
S’Rounding	For every subtype S of a floating point type T: S’Rounding denotes a function with the following specification: **function** S'Rounding (X : T) **return** T The function yields the integral value nearest to X, rounding away from zero if X lies exactly halfway between two integers. A zero result has the sign of X when S’Signed_Zeros is True. See A.5.3.
S’Safe_First	For every subtype S of a floating point type T: Yields the lower bound of the safe range (see 3.5.7) of the type T. If the Numerics Annex is not supported, the value of this attribute is implementation defined; see G.2.2 for the definition that applies to implementations supporting the Numerics Annex. The value of this attribute is of the type universal_real. See A.5.3.
S’Safe_Last	For every subtype S of a floating point type T: Yields the upper bound of the safe range (see 3.5.7) of the type T. If the Numerics Annex is not supported, the value of this attribute is implementation defined; see G.2.2 for the definition that applies to implementations supporting the Numerics Annex. The value of this attribute is of the type universal_real. See A.5.3.
S’Scale	For every decimal fixed point subtype S: S’Scale denotes the scale of the subtype S, defined as the value N such that S’Delta = 10.0**(–N). The scale indicates the position of the point relative to the rightmost significant digits of values of subtype S. The value of this attribute is of the type universal_integer. See 3.5.10.
S’Scaling	For every subtype S of a floating point type T: S’Scaling denotes a function with the following specification: **function** S'Scaling (X : T; Adjustment : universal_integer) **return** T Let v be the value X·T’Machine_Radix**(Adjustment). If v is a machine number of the type T, or if $\lvert v \rvert^3$T’Model_Small, the function yields v; otherwise, it yields either one of the machine numbers of the type T adjacent to v. Constraint_Error is optionally raised if v is outside the base range of S. A zero result has the sign of X when S’Signed_Zeros is True. See A.5.3.
S’Signed_Zeros	For every subtype S of a floating point type T: Yields the value True if the hardware representation for the type T has the capability of representing both positively and negatively signed zeros, these being generated and used by the predefined operations of the type T as specified in IEC 559:1989; yields the value False otherwise. The value of this attribute is of the predefined type Boolean. See A.5.3.
S’Size	For every subtype S: If S is definite, denotes the size (in bits) that the implementation would choose for the following objects of subtype S: A record component of subtype S when the record type is packed. The formal parameter of an instance of Unchecked_Conversion that converts from subtype S to some other subtype. If S is indefinite, the meaning is implementation defined. The value of this attribute is of the type universal_integer. See 13.3.
X’Size	For a prefix X that denotes an object: Denotes the size in bits of the representation of the object. The value of this attribute is of the type universal_integer. See 13.3.
S’Small	For every fixed point subtype S: S’Small denotes the small of the type of S. The value of this attribute is of the type universal_real. See 3.5.10.
S’Storage_Pool	For every access subtype S: Denotes the storage pool of the type of S. The type of this attribute is Root_Storage_Pool’Class. See 13.11.
S’Storage_Size	For every access subtype S: Yields the result of calling Storage_Size(S’Storage_Pool), which is intended to be a measure of the number of storage elements reserved for the pool. The type of this attribute is universal_integer. See 13.11.

T’Storage_Size	For a prefix T that denotes a task object (after any implicit dereference): Denotes the number of storage elements reserved for the task. The value of this attribute is of the type universal_integer. The Storage_Size includes the size of the task’s stack, if any. The language does not specify whether or not it includes other storage associated with the task (such as the “task control block” used by some implementations.) See 13.3.
S’Succ	For every scalar subtype S: S’Succ denotes a function with the following specification: **function** S'Succ(Arg : S'Base) **return** S'Base For an enumeration type, the function returns the value whose position number is one more than that of the value of Arg; Constraint_Error is raised if there is no such value of the type. For an integer type, the function returns the result of adding one to the value of Arg. For a fixed point type, the function returns the result of adding small to the value of Arg. For a floating point type, the function returns the machine number (as defined in 3.5.7) immediately above the value of Arg; Constraint_Error is raised if there is no such machine number. See 3.5.
S’Tag	For every subtype S of a tagged type T (specific or class-wide): S’Tag denotes the tag of the type T (or if T is class-wide, the tag of the root type of the corresponding class). The value of this attribute is of type Tag. See 3.9.
X’Tag	For a prefix X that is of a class-wide tagged type (after any implicit dereference): X’Tag denotes the tag of X. The value of this attribute is of type Tag. See 3.9.
T’Terminated	For a prefix T that is of a task type (after any implicit dereference): Yields the value True if the task denoted by T is terminated, and False otherwise. The value of this attribute is of the predefined type Boolean. See 9.9.
S’Truncation	For every subtype S of a floating point type T: S’Truncation denotes a function with the following specification: **function** S'Truncation (X : T) **return** T The function yields the value éXù when X is negative, and ëXû otherwise. A zero result has the sign of X when S’Signed_Zeros is True. See A.5.3.
S’Unbiased_Rounding	For every subtype S of a floating point type T: S’Unbiased_Rounding denotes a function with the following specification: **function** S'Unbiased_Rounding (X : T) **return** T The **function** yields the integral value nearest to X, rounding toward the even integer if X lies exactly halfway between two integers. A zero result has the sign of X when S’Signed_Zeros is True. See A.5.3.
X’Unchecked_Access	For a prefix X that denotes an aliased view of an object: All rules and semantics that apply to X’Access (see 3.10.2) apply also to X’Unchecked_Access, except that, for the purposes of accessibility rules and checks, it is as if X were declared immediately within a library package. See 13.10.
S’Val	For every discrete subtype S: S’Val denotes a function with the following specification: **function** S'Val(Arg : *universal_integer*) **return** S'Base This function returns a value of the type of S whose position number equals the value of Arg. See 3.5.5.
X’Valid	For a prefix X that denotes a scalar object (after any implicit dereference): Yields True if and only if the object denoted by X is normal and has a valid representation. The value of this attribute is of the predefined type Boolean. See 13.9.2.
S’Value	For every scalar subtype S: S’Value denotes a function with the following specification: **function** S'Value(Arg : String) **return** S'Base This function returns a value given an image of the value as a String, ignoring any leading or trailing spaces.
P’Version	For a prefix P that statically denotes a program unit: Yields a value of the predefined type String that identifies the version of the compilation unit that contains the declaration of the program unit. See E.3.
S’Wide_Image	For every scalar subtype S: S’Wide_Image denotes a function with the following specification: **function** S'Wide_Image(Arg : S'Base) **return** Wide_String

The function returns an image of the value of Arg, that is, a sequence of characters representing the value in display form. See 3.5.

S'Wide_Value	For every scalar subtype S: S'Wide_Value denotes a function with the following specification: **function** S'Wide_Value(Arg : Wide_String) return S'Base This function returns a value given an image of the value as a Wide_String, ignoring any leading or trailing spaces. See 3.5.
S'Wide_Width	For every scalar subtype S: S'Wide_Width denotes the maximum length of a Wide_String returned by S'Wide_Image over all values of the subtype S. It denotes zero for a subtype that has a null range. Its type is universal_integer. See 3.5.
S'Width	For every scalar subtype S: S'Width denotes the maximum length of a String returned by S'Image over all values of the subtype S. It denotes zero for a subtype that has a null range. Its type is universal_integer. See 3.5.
S'Class'Write	For every subtype S'Class of a class-wide type T'Class: S'Class'Write denotes a procedure with the following specification: **procedure** S'Class'Write(Stream : **access** Ada.Streams.Root_Stream_Type'Class; Item : **in** T'Class) Dispatches to the subprogram denoted by the Write attribute of the specific type identified by the tag of Item.
S'Write	For every subtype S of a specific type T: S'Write denotes a procedure with the following specification: **procedure** S'Write (Stream : **access** Ada.Streams.Root_Stream_Type'Class; Item : **in** T) S'Write writes the value of Item to Stream. See 13.13.2.

Annex L Pragmas - Language-defined Compiler Directives

Pragmas are Ada compiler directives. The word pragma has the same root as the word, pragmatic. It orginates in a Greek word which, roughly translated, means “Do this.” Some pragmas affect the process of compilation. Others tell the compiler about what elements belong in the Run-time Environment (RTE), and others restrict or expand the role of of some language feature.

pragma	All_Calls_Remote[(library_unit_name)];	— See E.2.3.
pragma	Asynchronous(local_name);	— See E.4.1.
pragma	Atomic(local_name);	— See C.6.
pragma	Atomic_Components(array_local_name);	— See C.6.
pragma	Attach_Handler(handler_name, expression);	— See C.3.1.
pragma	Controlled(first_subtype_local_name);	— See 13.11.3.
pragma	Convention([Convention =>] convention_identifier,[Entity =>] local_name);	— See B.1.
pragma	Discard_Names[([On =>] local_name)];	— See C.5.
pragma	Elaborate(library_unit_name{, library_unit_name});	— See 10.2.1.
pragma	Elaborate_All(library_unit_name{, library_unit_name});	— See 10.2.1.
pragma	Elaborate_Body[(library_unit_name)];	— See 10.2.1.
pragma	Export([Convention =>] convention_identifier, [Entity =>] local_name [, [External_Name =>] string_expression] [, [Link_Name =>] string_expression]);	— See B.1.
pragma	Import([Convention =>] convention_identifier, [Entity =>] local_name [, [External_Name =>] string_expression] [, [Link_Name =>] string_expression]);	— See B.1.
pragma	Inline(name {, name});	— See 6.3.2.
pragma	Inspection_Point[(object_name {, object_name})];	— See H.3.2.
pragma	Interrupt_Handler(handler_name);	— See C.3.1.
pragma	Interrupt_Priority[(expression)];	— See D.1.
pragma	Linker_Options(string_expression);	— See B.1.
pragma	List(identifier);	— See 2.8.
pragma	Locking_Policy(policy_identifier);	— See D.3.
pragma	Normalize_Scalars;	— See H.1.
pragma	Optimize(identifier);	— See 2.8.
pragma	Pack(first_subtype_local_name);	— See 13.2.
pragma	Page;	— See 2.8.
pragma	Preelaborate[(library_unit_name)];	— See 10.2.1.
pragma	Priority(expression);	— See D.1.
pragma	Pure[(library_unit_name)];	— See 10.2.1.
pragma	Queuing_Policy(policy_identifier);	— See D.4.
pragma	Remote_Call_Interface[(library_unit_name)];	— See E.2.3.
pragma	Remote_Types[(library_unit_name)];	— See E.2.2.
pragma	Restrictions(restriction{, restriction});	— See 13.12.
pragma	Reviewable;	— See H.3.1.
pragma	Shared_Passive[(library_unit_name)];	— See E.2.1.
pragma	Storage_Size(expression);	— See 13.3.
pragma	Suppress(identifier [, [On =>] name]);	— See 11.5.
pragma	Task_Dispatching_Policy(policy_identifier);	— See D.2.2.
pragma	Volatile(local_name);	— See C.6.
pragma	Volatile_Components(array_local_name);	— See C.6.

Windows 95 and NT Console Package

This package can be used to format a window with colors, place a cursor wherever you wish, and create character-based graphics on a Windows 95 or Windows NT console screen. You can access all of the control characters, and you can print the characters defined in Annex A, package Ada.Characters.Latin_1. This package is required form implementing the tasking problems shown in this book.

```
--------------------------------------------------------------------------------
--
-- File:        nt_console.ads
-- Description: Win95/NT console support
-- Rev:         0.1
-- Date:        18-jan-1998
-- Author:      Jerry van Dijk
-- Mail:        jdijk@acm.org
--
-- Copyright (c) Jerry van Dijk, 1997, 1998
-- Billie Holidaystraat 28
-- 2324 LK  LEIDEN
-- THE NETHERLANDS
-- tel int + 31 71 531 43 65
--
-- Permission granted to use for any purpose, provided this copyright
-- remains attached and unmodified.
--
-- THIS SOFTWARE IS PROVIDED ``AS IS'' AND WITHOUT ANY EXPRESS OR
-- IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED
-- WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
--
--------------------------------------------------------------------------------

package NT_Console is

   ----------------------
   -- TYPE DEFINITIONS --
   ----------------------

   subtype X_Pos is Natural range 0 .. 79;
   subtype Y_Pos is Natural range 0 .. 24;

   type Color_Type is (Black, Blue, Green, Cyan, Red, Magenta, Brown, Gray,
                       Light_Blue, Light_Green, Light_Cyan, Light_Red,
                       Light_Magenta, Yellow, White);

   --------------------
   -- CURSOR CONTROL --
   --------------------

   function Where_X return X_Pos;
   function Where_Y return Y_Pos;

   procedure Goto_XY (X : in X_Pos := X_Pos'First;
                      Y : in Y_Pos := Y_Pos'First);

   -------------------
   -- COLOR CONTROL --
   -------------------

   function Get_Foreground return Color_Type;
   function Get_Background return Color_Type;

   procedure Set_Foreground (Color : in Color_Type := Gray);
   procedure Set_Background (Color : in Color_Type := Black);

   --------------------
   -- SCREEN CONTROL --
```

```
   --------------------

   procedure Clear_Screen (Color : in Color_Type := Black);

   -------------------
   -- SOUND CONTROL --
   -------------------
   procedure Bleep;

   -------------------
   -- INPUT CONTROL --
   -------------------

   function Get_Key return Character;
   function Key_Available return Boolean;

----------------------
   -- EXTENDED PC KEYS --    Provides access to upper eight bit scan-code on a PC
----------------------
```

This is a list of special function keys available in Microsoft Operating Systems. The full list is in the package specification but we do not include here since it is seldom used.

Each keypress on a standard PC keyboard generates a scan-code. The scan-code is contained in an eight bit format that uniquely identifies the format of the keystroke. The scan code is interpreted by the combination of press and release of a keystroke. The PC's ROM-BIOS sees an Interrupt 9 which triggers the call of an interrupt handling routine. The Interrupt handling routine reads Port 96 (Hex 60) to decide what keyboard action took place. The interrupt handler returns a 2 byte code to the BIO where a keyboard service routine examines low-order and high order bytes of a sixteen bit value. The scan code is in the high-order byte.

Certain scan code actions are buffered in a FIFO queue for reading by some application program. Others trigger some immediate action such as reboot instead of inserting them into the queue.

The special keys in this list are those that can be queued rather than those that trigger an immediate operating system action.

C. Bibliography

Books Related to Ada

Ada 95 - The Language Reference Manual ANSI/ISO/IEC 8652:1995

Ada 95 Rationale, The Language and Standard Libraries, Ada Joint Program Office (with Intermetrics)

Beidler, John, *Data Structures and Algorithms, An Object-Oriented Approach Using Ada 95*, Springer-Verlag 1997, New York, ISBN 0-387-94834-1

Barnes, John G. P., *Programming in Ada 95*, Addison-Wesley, 1998, Second Edition
Be sure you get the second edition; many improvements over the first edition

Ben-Ari, Moti, *Understanding Programming Languages*, John Wiley & Sons, 1996

Ben-Ari, Moti, *Ada for Professional Software Engineers*, John Wiley & Sons, 1998

Booch, Grady, Doug Bryan, Charles Petersen, *Software Engineering with Ada*, Third Edition
Benjamin/Cummings, 1994 (Ada 83 only)

Booch, Grady, *Object Solutions, Managing the Object-Oriented Project*, Addison-Wesley, 1996

Burns, Alan; Wellings, Andy; *Concurrency in Ada*, Cambridge University Press, 1995

Burns, Alan; Wellings, Andy; *Real-Time Systems and Programming Languages*, Addison-Wesley, 1997

Bryan, Doulass & Mendal, Geoffrey, *Exploring Ada* (2 vols), Prentice-Hall, 1992

Cohen, Norman, *Ada As A Second Language*, , Second Edition, McGraw-Hill, 1996

Coleman, Derek, et al *Object-Oriented Development; The Fusion Method*, Prentice-Hall, 1994

Culwin, Fintan, *Ada, A Developmental Approach*, , Second Edition, 1997, Prentice-Hall

English, John, *Ada 95, The Craft of Object-Oriented Programming*, Prentice-Hall, 1997
(Now available for FTP download on the World Wide Web)

Fayad, Mohammed; Schmidt, Douglas; "Object-Oriented Application Frameworks", *Communications of the ACM*, October 1997 (Frameworks theme issue of CACM)

Feldman, Michael, *Software Construction and Data Structures with Ada 95*, Addison-Wesley, 1997

Feldman, M.B, and E.B. Koffman, *Ada95: Problem Solving and Program Design*, Addison-Wesley, 1996

Finklestein A. and Fuks S. (1989) "Multi-party Specification", *Proceedings of 5th International Workshop on Software Specification and Design*, Pittsburgh, PA , pp 185-95

Fowler, Martin and Kendall Scott, *UML Distilled*, Addison-Wesley Longman, 1997

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John; *Design Patterns, Elements of Resuable Object-Oriented Software*, Addison-Wesley, 1995

Gonzalez, Dean , *Ada Programmer's Handbook*, Benjamin/Cummings, 1993 (Ada 83 version only)

Jacobson, Ivar, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, 1994

Johnston, Simon, *Ada 95 for C and C++ Programmers*, Addison-Wesley, 1997

Kain, Richard Y., *Computer Architecture*, Prentice-Hall, 1989 (because software examples are in Ada)

Loftus, Chris (editor), *Ada Yearbook - 1994*, IOS Press, 1994

Meyer, Bertrand, *Object-Oriented Software Construction*, 2nd Editon, Prentice-Hall PTR, 1997

Naiditch, David, *Rendezvous with Ada 95*, John Wiley & Sons, 1995 (0-471-01276-9)

Rosen, Jean Pierre, ……………………….. HOOD …………………..

Pressman, Roger, *Software Engineering, A Practitioner's Approach, Fourth Edition*, McGraw-Hill, 1997

Salus, Peter H, *Handbook of Programming Languages, Vol 1, Object-Oriented Programming Languages*, MacMillan Technical Publishing, 1998 , ISBN 1-57870-009-4

Sigfried, Stefan, *Understanding Object-Oriented Software Engineering*, IEEE Press, 1995

Skansholm, Jan, *Ada From The Beginning*, , Third Edition, Addison-Wesley, 1997

Smith, Michael A., *Object-Oriented Software in Ada 95*, Thomson Computer Press, 1996

Sommerville, Ian, Software Engineering, Addison-Wesley, 1992 (an Ada friendly book on this topic)

Stroustrop, Bjarne, *The C++ Programming Language*, 3rd Edition, Addison-Wesley, 1997

Szyperski, Clemens,
Component Software: Beyond Object-Oriented Programming, Addison-Wesley, 1998

Taylor, David A, *Object-Oriented Technology; A Manager's Guide*, Addison-Wesley, 1992

Wheeler, David, A, Ada 95, *The Lovelace Tutorial*, Springer-Verlag, New York, 1997
[Other Books to be added]

Recommended Periodicals & Other Current Information

Most popular programmer's periodicals are staffed by editors who have little knowledge o interest in software engineering. Those who do have the knowledge and interest are woefully ignorant about Ada. Some of this ignorance stems from the general ignorance in the software community about Ada. Some of the following periodicals are listed for their general interest rather than their attention to serious software issues.

Ada Letters, A Bimonthly Publication of SIGAda, the ACM Special Interest Group on Ada
(ISSN 1094-3641)
A good source of accurate information regarding Ada

JOOP, Journal of Object-Oriented Programming, SIGS Publications,
Publishes articles and columns with positive perspective on Ada

C++ Report, (especially the Column, Obfuscated C++), SIGS Publications
If you want to be frightened about just how dangerous C++ really is, go to this source!

Embedded Systems Programming, Miller-Freeman Publications
Good Ada articles from time to time. Other good articles of interest to Ada practitioners

Dr. Dobbs Journal, Miller-Freeman
Generally misinformed about Ada. Editors, however, are open-minded about learning more accurate information

Internet Usenet Forum: comp.lang.ada

Internet Ada Advocacy ListServe: team-ada@acm.org

Internet AdaWorks Web Site: http://www.adaworks.com

Internet Ada Resources Association Web Site: http://www.adapower.com

Microsoft Windows Programming in Ada.
There are several good options. The easiest to learn is JEWL from
John English. The FTP is: ftp://ftp.brighton.ac.uk/pub/je/jewl/.

A commercial library, for serious Windows developers is CLAW from RR Software. This has a price tag but is worth every penny if you need industrial strength Ada Windows programs.
http://www.rrsoftware.com

The adapower.com site lists other options for those who want to program in Windows

Index

Error! No index entries found.