Workbook on Ada Programming Version 1.0

Author
Richard Conn
University of Cincinnati
Department of Electrical and Computer Engineering

March, 1992

Table of Contents

The purpose of this workbook is to teach you about Ada in an overview fashion and is by no means a complete tutorial. This workbook is provided as an aid to understanding the Ada language through numerous examples and is not intended as a replacement for a good textbook on Ada. The serious Ada student should acquire one or more of the texts referenced in the appendix to this workbook.

1. Overview and Basics

1.1. A Sample Ada Program

```
with Text_IO;  -- context specification
procedure Count_Down is  -- an Ada mainline is always a procedure
  Number_of_Iterations : Integer := 1;
    -- local variable definition
  package Int_IO is new Text_IO.Integer_IO (Integer);
    -- local package for Integer I/O
begin
  Text_IO.Put ("Enter number of iterations: ");
  Int_IO.Get (Number_of_Iterations);
  Text_IO.New_Line;
  Text_IO.Put ("The loop will run for ");
  Int_IO.Put(Number_of_Iterations);
  Text_IO.Put_Line (" iterations");
  for Index in 1 .. Number_of_Iterations loop
    Text_IO.Put ("Iteration: ");
    Int_IO.Put (Index, 4);  -- 4 column field
    Text_IO.New_Line;
  end loop;
end Count_Down;
```

A few notes about this program:

- Comments are denoted by a double-dash (--) and extend to the end of the line.

- The mainline is a procedure subprogram with no arguments. A subprogram is one of the four program units in Ada.

- The context specification indicates other Ada program units which are required by the current program unit. This procedure uses the package Text_IO for input and output support. Text_IO is an Ada package (a package is another kind of program unit) which contains many routines for inputting and outputting characters, strings, integers, etc.

- Number_of_Iterations is a variable of type Integer. The predefined type Integer is defined in the package called Standard, and Standard is automatically included in the context of all Ada program units so it need not be specified in a with statement.

- Int_IO is a package within this procedure that is created from a generic package called Integer_IO which is contained within the predefined package Text_IO. A generic unit (generic units may be subprograms or packages) is a third kind of Ada program unit.

l Most of the code in this program revolves around displaying and inputting text and integers. Note that the output is viewed as a stream, and the Put function places something into that stream. A new line is never assumed and must be explicitly placed into the stream via a Text_IO.New_Line procedure call or a Text_IO.Put_Line procedure call.

l Variables must be declared in Ada before they are used, but variables in a for loop are automatically created. The type of these variables is determined from the types of the parameters used. In this case, Index is of type Integer because Number_of_Iterations is of type Integer.

Problem 1.1: Type in the above program, compile it, and run it. Try giving various integer values, and be sure to include negative values. Try giving it erroneous inputs also, such as a character other than an integer.

1.2. Lexical Elements

There are 4 kinds of lexical elements, the basic language components of Ada:

l Identifiers -- names that identify various Ada objects, such as variables, constants, procedures, functions, and so forth. Identifiers begin with a letter and contain letters, digits, and underscore characters. Examples:

A Number_of_Iterations AGE P23 This_is_a_very_long_identifier

There are seven rules to follow when constructing Ada identifiers:

1. Identifiers can only contain letters, digits, and underscores.
2. Identifiers can be of any length as long as the entire identifier appears within a single line.
3. Ada does not distinguish between upper- and lower-case letters.
4. Identifiers cannot have the same name as a reserved word.
5. Identifiers must begin with a letter.
6. Identifiers cannot have underscores at the beginning, at the end, or adjacent to each other.
7. Underscores in identifiers are significant.

l Delimiters -- symbols that have a special meaning within Ada. They are often used as operators and statement terminators, but there are other purposes as well.

The single-character delimiters are:

+ - * / = < > & | ; # ( ) . : " '

The double-character delimiters are:

** /= <= >= := <> => << >> ..

l Literal -- a particular value of a type that is explicitly written and not represented by an identifier. Kinds of literals are:

- Integer literals: 0 187 2_546 2e3 12E+15 2#0010# 16#fc#
- Real literals: 0.0 127.021 4_728.001_002 16#f.2c# 1.2e35
- Character literals: 'a' 'A' ''' '8' '*'
- String literals: "This is a string" """" """Hello, Jack""", he said"

l Comment -- a string of characters beginning with a double dash and extending to the end of the line. Comments are ignored during compilation.

Problem 1.2: Write an Ada program that displays a single quote and a double quote.

1.3. Ada Program Units

An Ada source file contains one or more **Ada program units**. An Ada Program or System is composed of one or more program units, where a program unit is a subprogram, a package, a task, or a generic unit.

Each program unit is divided into two parts: (1) a **specification**, which defines its interface to the outside world, and (2) a **body**, which contains the code of the program unit.

When an Ada compiler runs, it compiles an Ada source file. The compiler places the program units from the source file into an **Ada Program Unit Library** if they compile successfully. Later, when an Ada linker (also commonly called a **binder**) is run, and executable is produced from program units within an Ada Program Unit Library. The mainline is specified to the linker, and the linker sets this program unit up to begin execution when the program is run. An Ada mainline is always an Ada procedure (a subprogram unit). Program units may be nested -- they can contain zero or more other program units.

Program Units: Subprogram. A subprogram is an expression of sequential action. Two kinds of subprograms exist: a procedure and a function.

An example of a procedure subprogram:

```
type REAL_ARRAY is array (1..20) of FLOAT;
procedure SORT (Items : in out REAL_ARRAY);  -- specification
procedure SORT (Items : in out REAL_ARRAY) is -- body
  -- definitions of types, objects, exceptions, and other program units
  -- (subprograms, packages, tasks, and generic units) local to SORT
begin
  -- body of code which implements SORT
  null;  -- no code for now
end SORT;
```

An example of a function subprogram:

```
function Is_Zero (Item : in FLOAT) return BOOLEAN;  -- specification
function Is_Zero (Item : in FLOAT) return BOOLEAN is -- body
  -- definitions of types, objects, exceptions, and other program units
  -- local to Is_Zero
begin
  null;  -- no code for now
  return TRUE;
end Is_Zero;
```

Program Units: Package. A package is a collection of computational resources, including data types, data objects, exception declarations, and other program units (subprograms, tasks, packages, and generic units). A package is a group of related items.

In object-oriented programming, a package contains the definition of a particular object or a class of objects. This includes the member data and all methods associated with the object or class.

Packages are fundamental to Ada. For instance, Ada by itself has no Input or Output capabilities, so the package Text_IO is provided with all Ada compilers to provide input and output capabilities for characters, strings, floating point numbers, fixed point numbers, integers, and user-defined types.

An example of an Ada package specification is:

```
package Console_Terminal_Screen is
  -- definitions of types, objects, exceptions, and other program units
  -- provided to external program units by this package
  subtype ROW is INTEGER range 1..24;
```

```
  subtype COLUMN is INTEGER range 1..80;
  procedure Clear_Screen;
  procedure Position_Cursor (At_Row : in ROW; At_Column : in COLUMN);
  procedure Write (Item : in STRING);
end Console_Terminal_Screen;
```

An example of an Ada package body which goes with this specification is:

```
package body Console_Terminal_Screen is
  -- definitions of types, objects, exceptions, and other program units used locally
  -- by this package
  procedure Clear_Screen is
  begin
    -- code to implement Clear_Screen
  end Clear_Screen;

  procedure Position_Cursor (At_Row : in ROW; At_Column : in COLUMN) is
  begin
    -- code to implement Position_Cursor
  end Position_Cursor;

  procedure Write (Item : in STRING) is
  begin
    -- code to implement Write
  end Write;

end Console_Terminal_Screen;
```

Program Units: Task. A task is an expression of action implemented in parallel with other tasks. A task is a body of code which may be implemented on one processor, a multiprocessor (more than one CPU), or a network of processors. Like other program units, a task has a specification and a body.

An example of a task specification is:

```
task Terminal_Driver is
  -- entry points to the task specify how other tasks communicate
  -- with this task
  entry Get (Char : out CHARACTER);
  entry Put (Char : in CHARACTER);
end Terminal_Driver;
```

An example of a task body with matches the above specification is:

```
task body Terminal_Driver is
  -- local data, etc.
begin
  accept Get (Char : out CHARACTER) do
    -- code which implements the Get entry
  end Get;
  accept Put (Char : in CHARACTER) do
    -- code which implements the Put entry
  end Put;
end Terminal_Driver;
```

Program Units: Generic Units. A generic unit is a special implementation of a subprogram or package which defines a commonly-used algorithm in data-independent terms. Generic units are reusable software

components which contain algorithm definitions.

An example of a generic subprogram specification and body is:

```
generic -- specification
  type ELEMENT is private;  -- thing manipulated
procedure Exchange (Item1 : in out ELEMENT; Item2 : in out ELEMENT);

procedure Exchange (Item1 : in out ELEMENT; Item2 : in out ELEMENT) is
  -- body
  Temp : ELEMENT;
begin
  Temp := Item1;
  Item1 := Item2;
  Item2 := Temp;
end Exchange;
```

Procedures as Main Programs. Ada does not have a separate construct for a main program. Instead, Ada program units (subprograms, packages, tasks, and generic units) are compiled into an Ada Program Unit Library and then, at some later time, one of the procedures is selected to be the mainline procedure at which execution of the program is to begin. A mainline procedure has no parameters.
µ §

The Ada Program Unit Library. The Ada compiler outputs into the current Ada Program Unit Library as its only target. The compiler does not necessarily create .o files like a C++ compiler may. Reuse is accomplished by accessing existing Ada program unit libraries, thereby gaining use of the program units contained in them.

All Ada compilers provide a common program unit library that contains the following packages:

- package STANDARD -- contains integers, floats, and operations on them
- package TEXT_IO -- support for I/O
- package SYSTEM -- ability to address memory
- package SEQUENTIAL_IO -- support for sequential I/O only
- package DIRECT_IO -- support for random I/O only
- package IO_EXCEPTIONS -- errors which may come up with I/O
- package LOW_LEVEL_IO -- special platform-specific I/O

Creating an Executable. The Ada Binder builds an executable from a procedure that is located any where in the Ada Program Unit Libraries. A chain of program units is assembled to create this executable:

- The mainline procedure comes first, and this procedure requires certain program units for support (it *depends* upon these program units and they are named in its **with** statements).
- The program units **withed** by the mainline procedure are incorporated into the executable, and these program units may **with** other program units.
- The next layer of program units is loaded, and so on as they **with** yet other program units.
- The Ada runtime system, which supports initialization, exception handling, tasking, and other features of the language, may either be included in the executable or tied into by the executable.

The **with** statement in Ada causes one program unit to gain access to another.

µ §

Problem 1.3: Write an Ada program that demonstrates a package, a procedure, and a function.

1.4. Basic and Extended Character Sets

The Basic Character Set (BCS) is one of two character sets used by Ada programs. The BCS was designed to facilitate transportability between computer systems. The BCS consists of:

- uppercase letters only: A - Z
- digits: 0 - 9
- special characters: " # ' ( ) * + - / , . : ; < = > _ | &
- the space character

The Extended Character Set (ECS) maps to the 95-character ASCII (American Standard Code for Information Interchange) set:

- all characters in the BCS
- more special characters: ! ~ $? @ [ ] { } \ ` ^ %
- lowercase letters: a - z

Package ASCII. Within the supplied package STANDARD is another package, **package ASCII**. Package ASCII provides two things: (1) names for the non-printing ASCII characters and (2) names for the characters in the ECS which are not a part of the BCS. Examples:

```
c1  : character := ASCII.NUL;
c2a : character := '#';
c2b : character := ASCII.SHARP;  -- same as c2a
c3a : character := 'a';
c3b : character := ASCII.LC_A;  -- same as c3a
```

The predefined package STANDARD is the only Ada package which is automatically withed by every Ada program unit without the programmer having to explicitly do so. Consequently, package ASCII is always available. A sample program:

```
with Text_IO;  -- for output
procedure ASCII_Demo is
begin
  Text_IO.Put("Ring the Bell: ");  -- Put for a string
  for I in 1 .. 20 loop
    Text_IO.Put (ASCII.BEL);  -- Put for a character
  end loop;
  Text_IO.New_Line;
end ASCII_Demo;
```

Problem 1.4: Write an Ada program that displays the numeric values of the ASCII characters from ASCII.NUL to ASCII.SUB.

1.5. Reserved Words

Reserved words are identifiers which may be used in only certain contexts. They may not be used as variables, enumeration literals, procedure names, etc. They may be a part of strings (e.g., "my package is in"). They may be a part of other lexical units (e.g., PACKAGE_52 is OK).

Complete List of Ada Reserved Words

abort	declare	generic	of	select
abs	delay	goto	or	separate
accept	delta		others	subtype
access	digits	if	out	
all	do	in		task
and		is	package	terminate
array	else		pragma	then
at	elsif	limited	private	type
	end	loop	procedure	
begin	entry			use
body	exception	mod	raise	
	exit		range	when
case		new	record	while
constant	for	not	rem	with
	function	null	renames	
			return	xor

reverse

Problem 1.5: Write an Ada program which uses the reserved words **elsif**, **for**, and **in**.

1.6. Package STANDARD

As mentioned above, package STANDARD is automatically withed and used by all Ada program units. Package STANDARD contains:

- type BOOLEAN and the associated operations:
  = /= < <= > >= and or xor not
- type INTEGER and the associated operations:
  = /= < <= > >=
  + - abs (unary operations)
  + - * / rem mod (binary operations)
  **
- type FLOAT and the associated operations:
  = /= < <= > >=
  + - abs (unary operations)
  + - * / (binary operations)
  ** (INTEGER exponent)
- the types universal real, universal integer, and universal fixed along with their operations:
  universal real := universal integer * universal real
  universal real := universal real * universal integer
  universal real := universal real / universal integer
  universal fixed := user-defined fixed * user-defined fixed
  universal fixed := user-defined fixed / user-defined fixed
- types CHARACTER and STRING and their associated operations:
  = /= < <= > >=
  & (binary operation for both CHARACTER and STRING)
- subtype NATURAL is INTEGER range 0 .. INTEGER'LAST;
- subtype POSITIVE is INTEGER range 1 .. INTEGER'LAST;
- predefined exceptions:
  CONSTRAINT_ERROR
  PROGRAM_ERROR
  TASKING_ERROR
  NUMERIC_ERROR
  STORAGE_ERROR

Problem 1.6: Write an Ada program which builds and displays a truth table showing the results of the logical and, or, and xor of two Boolean variables.

2. Types

In Ada, every object belongs to a class of objects. Classes of objects are denoted by a **type**. A type characterizes:

- a set of values which objects of that type may take on (e.g., integers range in value from INTEGER'FIRST to INTEGER'LAST)
- a set of attributes (e.g., INTEGER'LAST is the last integer)
- a set of operations which may be performed on objects of that type (e.g., the package STANDARD defines the type INTEGER and the operations which may be performed on objects of type INTEGER)

Several classes of types are available in the Ada language:

- scalar data types, which assume a single value at any one time
  - integer
  - real (both floating point and fixed point)
  - enumeration
- composite data types, which assume one or more values in various parts of them at any one time
  - array
  - record
- access data types, which are pointers to other types of objects (these are handled in a distinctly different fashion from scalar data types, although they do assume a single value at any one time)
- private data types, whose contents are not of interest to the user (employed in conjunction with Ada packages)
- subtypes, which are created from parent types and are compatible with the parent types
- derived types, which are created from parent types and are not compatible with the parent types

This section of the workbook discusses all of these types except for private data types, which are covered in the discussion on Ada packages.

2.1. Scalar Data Types

Scalar types are types that do not have any lower-level components; scalar types assume a single value at any one time, like an integer.

There are two kinds of scalar types: discrete and real. A discrete type is one whose values have immediate successors and predecessors, such as Boolean, Integers, and user-defined enumeration types. A real type is one whose values are numbers that form a continuum, such as floating point numbers and fixed point numbers.

Scalar types are either predefined or user-defined. The following table presents the predefined scalar data types and the syntax for defining user-defined scalar data types:

Kind of Scalar Data Type	***Explanation***
l Integer:	
INTEGER	a predefined type
NATURAL	a predefined type, >= 0
POSITIVE	a predefined type, >= 1
type INDEX is range 1..50;	syntax for creating user-defined integer-like types
l Real (floating point and fixed point):	
FLOAT	a predefined type
type MASS is digits 10;	a user-defined floating point type with 10 significant digits
type VOLTAGE is delta 0.01 range 0.0 .. 50.0;	a user-defined fixed point type
l Enumeration:	
BOOLEAN	a predefined type, values are FALSE and TRUE
CHARACTER	a predefined type, values are from the Extended Character Set
type COLOR is (RED, GREEN, BLUE);	a user-defined enumeration type

It is important to be able to distinguish between numeric and discrete scalar data types since only discrete types may be used for loop variables, such as in a **for** loop.

Numeric and Discrete Types

	Integer	***Real***	***Enumeration***
Numeric	X	X	
Discrete	X		X

Universal types, which are of a literal form and may match to any of a number of similar types, exist in Ada. The following classes of universal types exist:

- Universal Integer
  - Integer Literals, e.g.: 12
  - Integer Named Numbers, e.g.: DOZEN : constant := 12;
- Universal Real
  - Real Literals, e.g.: 3.14159
  - Real Named Numbers, e.g.: PI : constant := 3.14159_26535;

In clarification, do not confuse these universal values with typed values, such as:

```
DOZEN : constant INTEGER := 12;  -- this is of type INTEGER
DOZEN : constant := 12;          -- this is a universal integer
```

There are two main advantages to universal types:

- Universal types do not have any practical size constraints:

SPEED_OF_LIGHT : constant := 186_282; -- valid on even 16-bit machines

- Code may execute faster: when named numbers are combined with other named numbers or numeric literals, the resulting expression may be evaluated at compilation time rather than run time.

Expressions consisting of only named numbers or numeric literals are called *literal expressions*. Named numbers may be initialized to literal expressions but not to non-literal expressions:

```
DOZEN : constant := 12;
BAKERS_DOZEN : constant := DOZEN + 1;  -- OK because 'DOZEN+1' is
   -- a literal expression
DOZEN : constant INTEGER := 12;
BAKERS_DOZEN : constant := DOZEN + 1;  -- not OK because DOZEN is a
   -- variable
```

Problem 2.1: Write an Ada program which consists of a procedure inside a package that employs an enumeration type containing the values RED, GREEN, BLUE, YELLOW, ORANGE, TURQUOISE, MAGENTA, and VIOLET, in that order. The procedure should contain a for loop that displays the values from RED on up to some limit which is specified by a constant. Use typename'IMAGE to convert from an enumeration type value to a representative string.

2.2. Subtypes and Derived Types

Subtypes are types created from an existing "parent" type which are distinct but compatible with the parent. Objects of a subtype may be mixed with objects of the parent type in an expression. For example,

```
subtype SINT is INTEGER range 1..10;
I  : INTEGER;
SI : SINT;
SI := 5;
I  := 2 + SI;  -- OK because SI is a subtype
```

Derived types are types created from an existing "parent" type which are distinct and separate (incompatible) from the parent. They have the same physical structure but the compiler views them as being distinct and enforces this view by not allowing operations dealing with the parent and the derived types to exist unless they are defined explicitly by the programmer. For example:

```
type SINT is new INTEGER range 1..10;  -- this is derived
I  : INTEGER;
SI : SINT;
SI := 5;
I  := 10 + SI;  -- will raise an error message at compile time
```

Derived types are different from **subtypes**:

- A derived type introduces a new type, distinct from its parent.
- A subtype places a restriction on an existing type, but it is compatible with its parent.

Derived types make a lot of sense, providing a check when mapping to the real world. For instance, in the real world, you would never try to add something of type SPEED (say, in miles per hour) to something of type TIME (say, in hours). It simply does not make sense. Derived types in Ada prevent this kind of thing from happening:

```
type SPEED is new FLOAT range 0.0 .. 1_000_000.0;  -- MPH
type TIME is new FLOAT range 0.0 .. 24.0;  -- hours
S  : SPEED := 25.0;
T  : TIME := 12.00;
S  := S + 5.0;  -- legal, no problem
T  := T + 1.0;  -- legal, we are still within the range constraint
```

```
S   := S + T;   -- illegal, flagged at compile time
```

Problem 2.2: Write an Ada program which uses the definitions for SPEED and TIME from above. Add a new derived type for DISTANCE. Create an overloaded "*" operator which multiplies SPEED by TIME to produce DISTANCE. Use this operator to create a short table of speeds, times, and distances.

2.3. Arrays

An **array** is an object that consists of multiple homogenous components (i.e., each component is of the same type). An entire array is reference by a single identifier:

```
type FLOAT_ARRAY is array (1..10) of FLOAT;  -- array type declaration
My_Float_Array : FLOAT_ARRAY;  -- array object definition
```

Each component of an array is referenced by the identifier which references the array being followed by an index in parentheses. Notice that two types are generally involved: the type of the array elements and the type of the index. Continuing the above example:

```
My_Float_Array(5) := 12.2;  -- assign one element
My_Float_Array(5..7) := (1.0, 2.0, 3.0);
   -- assign three element using a slice
for I in My_Float_Array'FIRST .. My_Float_Array'LAST loop
   -- init all elements
  My_Float_Array(I) := 0.0;
end loop;
```

The general syntax of an array type declaration is:

type array_type_name is array (index_specification) of element_type;

Some notes:

- array_type_name is the name given to this type, not the name of a specific array; specific arrays are declared later as array objects
- index_specification is the type and value range limits, if any, of the index
- element_type is the type of the array elements

Some examples of array type declarations and array definitions:

```
type COLOR is (RED, GREEN, BLUE);  -- an enumeration type (used later)

type VALUES is array (1..8) of FLOAT;  -- a vector of 8 real numbers
My_Floats : VALUES;  -- object definition
His_Floats : VALUES := (1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8);
   -- object definition with initialization
Zero_Floats : VALUES := (others => 0.0);
   -- initialized to 0.0
Our_Floats : VALUES :=
  (RED   => 1.0,
   GREEN => 2.0,
   BLUE  => 3.0);  -- index associations spelled out for readability

type CVAL is array (COLOR) of FLOAT;
   -- a vector of 3 real numbers indexed by RED, GREEN, and BLUE
My_Color_Values : CVAL;  -- object definition
Reference_Color_Values : constant CVAL := (1.1, 2.2, 3.3);
   -- constant object definition
```

```
type SCREEN_DOTS is array (1..1024, 1..1024) of COLOR;
My_CRT_Screen : SCREEN_DOTS := (others => (others => RED));
   -- all RED
```

An entire array may be initialized by assigning it to an **array aggregate**. Some examples:

```
type MENU_SELECTION is (SPAM, MEAT_LOAF, HOT_DOG, BURGER);
   -- an enumeration naming the different items on the menu
type DAY is (MON, TUE, WED, THU, FRI);
   -- an enumeration naming the days of the week
type SPECIAL_LIST is array (DAY) of MENU_SELECTION;
   -- an array type, indexed by the days of the week, of menu items
Specials : SPECIAL_LIST;  -- an array

Specials := SPECIAL_LIST'(SPAM, HOT_DOG, BURGER, MEAT_LOAF, SPAM);
   -- an array aggregate assigning a value to the array Specials
Specials := (SPAM, HOT_DOG, BURGER, MEAT_LOAF, SPAM);
   -- the qualifier is not needed -- it simply improves readability
Specials := (MON => SPAM,
             TUE => HOT_DOG,
             WED => BURGER,
             THU => MEAT_LOAF,
             FRI => SPAM);  -- another improvement on readability
Specials := (MON | FRI => SPAM,
             TUE | WED | THU => BURGER);
Specials := (MON .. WED => BURGER, others => MEAT_LOAF);
```

Some additional notes on arrays:

- Arrays may have as many dimensions as desired.
- So far, array types have been *constrained* (i.e., the number of elements in the arrays have been determined in advance during the declaration of the array type). In Ada, array types may also be *unconstrained*, where the objects derived from these types are not constrained until the definitions of these objects. For example:

```
type FLOAT_ARRAY is array (NATURAL range <>) of FLOAT;
```

```
   -- index is of type NATURAL and is constrained at object definition
My_Array   : FLOAT_ARRAY (1..10);  -- 10 elements
His_Array  : FLOAT_ARRAY (5..12);  -- 8 elements
Zero_Array : constant FLOAT_ARRAY := (0.0, 0.0, 0.0);
   -- 3 elements indexed from NATURAL'FIRST (0, 1, 2)
```

- A **STRING** is an unconstrained array indexed by **POSITIVE** of **CHARACTER** objects. The type STRING is predefined in the package STANDARD:

```
type STRING is array (POSITIVE range <>) of CHARACTER;
```

- Once a STRING object has been defined, it may be assigned a value by using array aggregate notation or by using quotes:

```
My_Name : STRING(1..4) := "John";
```

```
My_Name := ('J', 'i', 'm', ' ');
```

Problem 2.3: Write an Ada program which uses the following types to create several tic-tac-toe board positions and display them:

```
type SQUARE is (EMPTY, CROSS, ZERO);
```

```
type ROW is (ROW_1, ROW_2, ROW_3);
type COL is (COL_1, COL_2, COL_3(;
type TIC_TAC_TOE_BOARD is array (ROW, COL) of SQUARE;
```

2.4. Strings

A *string* is a kind of vector used so often that it is useful to make a special mention of it. Some notes about strings follow.

- A double quote may be placed into a string by using two double quotes in a row:

```
Message : constant STRING := """Hi""";
```

- Strings are fixed in their length when they are defined. Ada does not support any predefined variable-length strings, but such strings can be implemented (see CS Parts).

```
Pet_Name : STRING(1..5);
```

```
Pet_Name := "Pete";  -- illegal because Pet_Name has 5 characters
                -- and "Pete" has 4
Pet_Name := "Repeat";  -- illegal for similar reasons
Pet_Name := "Pete ";   -- OK (note trailing space)
Pet_Name(1..4) := "Jake";  -- OK due to slice size matching string
                  -- length
```

- *Slices* can be used to assign parts of a string:

```
Message(1..2) := "Hi";
```

- Although the size of a STRING variable is set by its constraint, the size of a STRING constant may be inferred from the size of the aggregate assigned to it:

```
My_Pet : constant STRING := "Jake the Snake";
```

- **Package Text_IO** supports a STRING input procedure called Get_Line which may be used to input values into STRINGs. Get_Line requires the name of the STRING object and a variable of type NATURAL as parameters:

```
Input_Line : STRING(1..80);
```

```
Input_Last : NATURAL;
Text_IO.Get_Line (Input_Line, Input_Last);
  -- Assume that the user types "HELLO<CR>", where <CR> is the RETURN key
  -- Input_Line(1..5) = "HELLO" and Input_Last = 5 are the results
```

Problem 2.4: Write an Ada program which inputs five lines from the console using package Text_IO. After the five lines are input, the program displays them back to the user.

2.5. Boolean Vectors

A **boolean vector** is a user-defined type which is a vector of BOOLEAN objects:

```
type BOOLEAN_VECTOR is array (POSITIVE range <>) of BOOLEAN;
```

A boolean vector is the only type of array that can be operated on by the logical operators *and*, *or*, *xor*, and *not* as a whole (i.e., the operators operate on the entire array at one time rather than forcing the programmer to reference each element individually). For example:

```
declare
  type BOOLEAN_VECTOR is array (POSITIVE range <>) of BOOLEAN;
  T : constant BOOLEAN := True;    F : constant BOOLEAN := False;
  A : BOOLEAN_VECTOR (1..4) := (T, F, T, F);
  B : BOOLEAN_VECTOR (1..4) := (T, F, F, T);
  C : BOOLEAN_VECTOR (1..4);
begin
  C := not A;    -- yields (F, T, F, T);
  C := A and B;  -- yields (T, F, F, F);
  C := A or B;   -- yields (T, F, T, T);
  C := A xor B;  -- yields (F, F, T, T);
end;
```

Problem 2.5: Using type BOOLEAN_VECTOR, write an Ada program which generates truth tables for *and*, *or*, and *xor*.

2.6. Array Attributes and Operations

Ada allows **attributes** associated with various kinds of objects to be accessed in a general-purpose fashion by using predefined attribute names attached to object names with a tick (single quote). An array composite data type has four attributes of particular interest:

- FIRST - the value of the first index value
- LAST - the value of the last index value
- RANGE - array'FIRST .. array'LAST
- LENGTH - the number of elements in an array

For example:

```
type FA is array (INTEGER range <>) of FLOAT;
X : FA (2..5);
I : INTEGER;

I := X'FIRST;   -- I = 2
I := X'LAST;    -- I = 5
for J in X'RANGE loop  -- J goes from 2 to 5
  null;
end loop;
I := X'LENGTH;  -- I = 4
```

These attributes apply to array objects (which are, of course, constrained) and constrained array types. Operations on arrays are:

Operation	*Restriction*
Attributes (FIRST, etc.)	None
Logical (not, and, or, xor)	Arrays must be BOOLEAN vectors of the same length and type
Concatenation (&)	The arrays must be vectors (one-dimensional)
Assignment (:=)	The arrays must be of the same size and type
Type Conversions	The arrays must be of the same size and component and index types
Relational (<, >, <=, >=)	The arrays must be discrete vectors of the same type
Equality (=, /=)	The arrays must be of the same type

Problem 2.6: Write an Ada program which contains a procedure that displays the values of an array of floating point numbers passed to it. The procedure should use the array attributes to determine the index range of the array. Pass at least two arrays of different index ranges to this procedure.

2.7. Records

A **record** is a composite data type in which the components of the record are not necessarily homogenous (i.e., of the same type, as in an array). A record type may be composed of a float, a character string, and an integer, for example, where an array type may be composed of just integers or just floats. Records are often the basis for *abstract data types* in Ada, and *abstract data types* are the basis for the definition of *object member data* in an object-based design written in Ada.

The most basic kind of record is that declared without **discriminants** (parts of a record definition that allow one instance of a record to vary from another, thereby supporting *variant records*). The general syntax of a record type declaration without discriminants is:

```
type record_type_name is record
  record_components;
end record;
```

where record_components is of the form

```
  field_name : type_name [:= initial_value];
```

When an object of a record type is created, the fields of the object are accessed by using a dot notation of the form:

```
object_name.field_name
```

If a field is a record itself, the dot notation may be continued. Examples of a record type and object definition and assignments are:

```
type MY_RECORD is record
  I : INTEGER;
  F : FLOAT := 5.0;
end record;
X : MY_RECORD;  -- X.I is undefined, and X.F is initialized to 5.0
X : MY_RECORD := (I => 2);  -- X.I is initialized to 2, X.F is
                    -- initialized to 5.0
X : MY_RECORD := (I => 2, F => 4.0);  -- X.I = 2, X.F = 4.0
X : MY_RECORD := (2, 4.0);  -- same as above
```

Records with Discriminants. Record types with discriminants may be used to define records to be of the same type even though the kind, number, and size of the components differ between individual instances. *Variant records* are those that differ from one another in the kind and number of components. The variable component, which is just another component in the record, is specified in the type declaration like a parameter. An example of a variant record in Ada:

```
type RECORDING_MEDIUM is (PHONOGRAPH, CASSETTE, CD);
type MUSIC_TYPE is (CLASSICAL, JAZZ, NEW_AGE, FOLK, POP);
type RPM is (RPM_33, RPM_45);
type RECORDING (Device : RECORDING_MEDIUM := CD) is record
  Music : MUSIC_TYPE;
  case Device is
    when PHONOGRAPH =>
      Speed : RPM;
    when CASSETTE =>
      Length : NATURAL;
    when CD =>
      null;
  end case;
end record;
My_CD : RECORDING := (Music => POP);
His_Record : RECORDING (PHONOGRAPH) := (Music => JAZZ, Speed =>
RPM_33);
```

```
My_Tape : RECORDING(CASSETTE) := (Music => CLASSICAL, Length => 60);
```

Problem 2.7: Write an Ada program which manipulates an object of type BOOK, where the BOOK class of objects consists of a character string for the title of the book, a character string for the author, a floating point number for the price, and an integer for the publication year. Create some objects of type BOOK, initialize them, and display their values.

2.8. Access Types

Access types are used to declare variables (pointers) that access dynamically allocated variables. A dynamically allocated variable is brought into existence by an *allocator* (the keyword **new**). Dynamically allocated variables are referenced by an access variable, where the access variable "points" to the variable desired. The general form of an access type declaration is:

type access_type_name is access type_name;

A variable of an access type is the only kind of variable in Ada which is initialized without the programmer having to do anything. All variables of an access type are initialized to **null** (another keyword) when they are created.

With variables of an access type, the special suffix **.all** refers to the *value* of the variable being accessed. If the variable accessed is a record, the record fields may be addressed by using the same dot notation as used for a conventional record variable.

Examples:

```
type INTEGER_ACCESS_TYPE is access INTEGER;
type RECORD_TYPE is record
  I : INTEGER;
  F : FLOAT;
end record;
type RECORD_ACCESS_TYPE is access RECORD_TYPE;
P1, P2 : INTEGER_ACCESS_TYPE;
R1 : RECORD_ACCESS_TYPE;
P1 := new INTEGER;  P1.all := 5;
P2 := new INTEGER'(12);  -- allocate and initialize
R1 := new RECORD_TYPE;
R1.I := 12; R1.F := 2.2;
R1 := new RECORD_TYPE'(I => 10, F => 10.0);
```

Problem 2.8: Write an Ada program which manipulates objects of a record type that contains a NATURAL indicating the number of valid elements in an array of floating point numbers and an array of up to 100 floating point numbers. This program should contain a procedure which receives an access type that addresses one of these objects and prints out the valid numbers. Have the mainline call this procedure two or more times with different arguments.

2.9. Object Representation

There are several attributes associated with type declarations and object definitions in Ada which can be used to direct affect the internal implementation details of these types and objects. The following are attributes which may be applied to various entities in order to determine some of their specific properties:

Attribute	*Description*
ADDRESS	reports the memory location of an object, program unit, label, or task entry point
SIZE	reports the size, in bits, of an object, type, or subtype
STORAGE_SIZE	reports the amount of available storage for access types and tasks; if P is an access type, P'STORAGE_SIZE gives the amount of space required for an object accessed by P; if P is a task, P'STORAGE_SIZE give the number of storage units reserved for task activation
POSITION (records only)	reports the offset, in storage units, of a record component from the beginning of a record
FIRST_BIT (records only)	reports the number of bits that the first bit of a record component is offset from the beginning of the storage unit in which it is contained
LAST_BIT (records only)	reports the number of bits that the last bit of a record component is offset from the beginning of the storage unit that contains the first bit of the record component

These attributes only return information about the associated entities. In addition to these attributes, the following *representation clauses* (also known as *representation specifications* or *rep specs*) are available in Ada to set certain attributes:

- Length clauses -- establish the amount of storage used for objects

```
type DIRECTION is (UP, DOWN, RIGHT, LEFT);
for DIRECTION'SIZE use 2;  -- 2 bits
```

- Enumeration clauses -- specify the internal representation of enumeration literals

```
type BIT is (OFF, ON);
for BIT'SIZE use 1;  -- 1 bit
for BIT use (OFF => 0, ON => 1);
```

l Record Representation clauses -- associated record components with specific locations in bit fields

```
type STATUS_WORD is record
  Carry_Bit    : BIT;
  Overflow_Bit : BIT;
  Fill_1       : BIT;
  Fill_2       : BIT;
  Fill_3       : BIT;
  Fill_4       : BIT;
  Fill_5       : BIT;
  Zero_Bit     : BIT;
end record;
for STATUS_WORD'SIZE use 8;  -- 8 bits
for STATUS_WORD use record
  Carry_Bit    at 0 range 0..0;  -- bit 0
  Overflow_Bit at 0 range 1..1;  -- bit 1
  Fill_1       at 0 range 2..2;
  Fill_2       at 0 range 3..3;
  Fill_3       at 0 range 4..4;
  Fill_4       at 0 range 5..5;
  Fill_5       at 0 range 6..6;
  Zero_Bit     at 0 range 7..7;  -- bit 7
end record;
```

l Address clauses -- specify the addresses of objects

```
CPU_STATUS : STATUS_WORD;       -- define object
for CPU_STATUS use at 16#100#;  -- define address of object
```

Note that Address clauses may also be apply to task entry points to establish interrupt mappings:

```
task RUNNING_SCORE is
  entry HIT;  for HIT use at 16#020#;
  entry MISS; for MISS use at 16#040#;
end RUNNING_SCORE;
```

Problem 2.9: Write an Ada program which extends the definition of STATUS_WORD given above, including a field which is more than one BIT long, and sets the values of several STATUS_WORD objects. Display the values of these objects.

2.10. Operators

The following operators are supported in the Ada language:

Notes	*Operators*	*Precedence*
Exponentation, not and abs	** not abs	Highest

Multiply Operators	* / mod rem	
Unary Operators	+ -	
Binary Operators	+ - &	
Relational Operators	= /= < <= > >=	
Membership Operators	in not in	
Logical Operators	and or xor	
Short-Circuit Operators	and then or else	Lowest

Sample expression using these operators:

Expression	*Notes*
PI	a simple expression
(B**2) - (4.0 * A * C)	
B**2 - 4.0 * A * C	same meaning as the above
CH in 'a' .. 'z'	a boolean expression
24.2 ** 3	a static expression
(not SUNNY) or WARM	a boolean expression
not SUNNY or WARM	same meaning as the above
not (SUNNY or WARM)	different from above
"Hello" & " " & "Joe"	string concatenation
b > 0 and the a/b < 5	short circuit boolean expression

Problem 2.10: Write an Ada program which inputs a set of numbers and displays their quotients, using short circuit operators to check for potential errors before attempting the divides.

3. Ada Statements

A *statement* in Ada is a sequence of characters terminated by a semicolon (;). For example,

```
Value := Value + 1;  -- an assignment statement

Value
:=
2
;  -- another assignment statement

Value := 2;  -- same as the last assignment statement

if A < 4 then
  A := 4;
end if;  -- an if statement which contains one assignment statement
```

There are four broad categories of statements in Ada:

- Sequential Control Statements
  - Assignment
  - Block
  - Null
  - Return
  - Procedure Call
- Conditional Control Statements
  - Case
  - If
- Iterative Control Statements
  - Exit
  - Loop (including For Loop and While Loop)
- Other Statements
  - Abort
  - Accept
  - Code
  - Delay
  - Entry Call
  - Goto
  - Raise
  - Select

3.1. Sequential Control Statements

An *assignment statement* changes the value of a variable. The colon-equals operator (:=) is used to perform the assignment, as opposed to the equals operator (which is a test for equality, resulting in a boolean value of True or False). Examples of assignment statements:

```
Value := 1;
Value := SQRT(B**2 + A**2);
```

A *block statement* declares a logically-cohesive body of code with, optionally, its own local variables. In essence, a block is like an inline procedure with no parameters. The variables available within a block are all the variables in the subprogram which contains the block plus its own local variables. The general syntax of a block statement is:

```
label:  -- optional block label
declare  -- optional local variable declaration region
  -- local variables are defined here
begin -- marks the beginning of the code in the block
  null;  -- local statements
end label;  -- end of block
```

An example of a block is:

```
declare
  local_1 : INTEGER;
begin
  local_1 := 2;
  value := value / local+1;  -- the variable VALUE is global
                        -- to the block
end;
```

The *null statement* is simply the statement

null;

and it is usually employed as a placeholder during design to mark where code will go in the future while still allowing the design to be compiled.

The *return statement* is used to return from a subprogram. If the subprogram is a procedure, the form of the return statement is:

return;

and if the subprogram is a function, the form of the return statement is:

return value;

-- for example:

```
procedure X is
begin
  return;
end X;

function Y return FLOAT is
begin
  return 1.0;
end Y;
```

A *procedure call statement* is simply a call to a procedure. The procedure name may be prefixed with the name of a containing program unit, such as a package, if necessary. The general form of a procedure call:

```
procedure_name;  -- no arguments
procedure_name(arg, arg, ...);  -- arguments
package_name.procedure_name (arg, arg, ...);
   -- procedure in a package
```

Examples:

```
Text_IO.Put_Line("Hello, world");
```

```
Put("Enter Text: ");
Stacks.Push (100.0, My_Stack);
```

Problem 3.1: Write an Ada program which contains a procedure and a block with local variables. Place the procedure inside the block and call it several times.

3.2. Conditional Control Statements

There are two basic kinds of conditional control statements in Ada: the if statement and the case statement.

The *if statement* is of the following general form:

```
if boolean_expression then
  -- statements to execute if true
elsif another_boolean_expression then -- alternate if
  -- statements to execute if true
elsif yet_another_boolean_expression then -- as many as desired
  -- statements to execute if true
else -- what to do if all else fails
  -- statements to execute if all are false
end if;  -- end of if statement
```

Example of the if statement:

```
type COLOR is (RED, GREEN, YELLOW);
Stop_Light : COLOR := RED;
if Stop_Light = RED then
  Stop;  -- procedure call
elsif Stop_Light = YELLOW then
  Close_Eyes; Go_Fast;  -- two procedure calls
else  -- must be GREEN
  Stop;
  Look_Both_Ways;
  Go;  -- three procedure calls
end if;

if Value > 10 then
  Value := Value - 10;
end if;
```

The *case statement* is a multi-way selection. The general form of the case statement is:

```
case variable_name is
  when variable_value =>
    -- statements to perform when variable_name = variable_value
  when variable_value2 =>
    -- and so on
  when others =>
    -- statements to perform when variable_name
    -- takes on any other value
end case;
```

Every possible value of the case variable name must be covered in one and only one **when** clause. The **when others** clause is used to cover all instances not explicitly covered before and always appears as the

last **when** clause. Choices in a **when** clause must be static (able to be resolved at compile time). Case variable names must be *discrete*.

Examples of case statements:

```
case Value is
  when 1 | 3 | 5 | 7 | 9 => Kind := ODD;
  when others => Kind := EVEN;
end case;

case Value is
  when 0 .. 9 => Kind := LESS_THAN_10;
  when others => Kind := TEN_OR_MORE;
end case;

case Stop_Light is
  when RED =>
    Stop;
  when GREEN =>
    Look_Both_Ways;
    Go;
  when YELLOW =>
    Close_Eyes;
    Go_Fast;
  when others =>  -- off?  but we have already covered all
                  -- the enumeration values
    Stop;
    Look_Both_Ways;
    Go;
end case;
```

Problem 3.2: Write a function which acts on a CHARACTER and returns an enumeration value of the set (LOWER_CASE, UPPER_CASE, SHARP, COMMA, SEMICOLON, DIGIT, SOME_OTHER). Write an Ada mainline which calls this function with a number of different characters and displays the character and the result of the function call.

3.3. Iterative Control Statements

There are two kinds of *exit statements* and three kinds of *loop statements* which make up Ada's *iterative control statements*.

The *exit statements* cause a premature exit from a loop or a block. The two kinds of exit statements are:

```
exit;  -- unconditional
exit when boolean_expression;  -- conditional
```

Examples:

```
for I in 1..10 loop
  if I = 5 then
    exit;
  end if;
end loop;
```

```
for I in 1..10 loop
  exit when I=5;
end loop;
```

The *loop statements* cause iterations to occur. The three kinds of loop statements are:

- *Simple loop*:

```
optional_label:
loop
  statements;
end loop optional_label;
```

- *While loop*:

```
optional_label:
while boolean_expression loop
  statements;
end loop optional_label;
```

- *For loop*:

```
optional_label:
for loop_variable in loop_value_range loop
  statements;
end loop optional_label;
```

Examples of loop statements:

```
loop -- simple loop
  Bit := Status_Bit;
  exit when Bit = ON;
end loop;

while Status_Bit = OFF loop
  null;  -- nothing for now
end loop;

i := 42;
add_up:
for i in 1..20 loop  -- for loop, creates local I variable and
                     -- hides outer I variable
  sum := sum + i;
end loop add_up;
sum := sum + i;  -- add in outer I variable
```

Problem 3.3: Write an Ada program which loops up to 10 times, accepting a string input from the user and exiting if the string contains the word "EXIT" in the first four characters.

4. Ada Program Units

Program units in Ada are the basic components of Ada libraries. When an Ada compiler compiles an Ada source file, it places the programs units in this source file which it compiles successfully into a "current" program unit library. When an Ada system is completely compiled, an executable is built by a binder starting with a procedure program unit that has no parameters and including all the program units it requires, all the program units they require, and so on until no more program units are needed. An Ada runtime system or an interface to an Ada runtime system is added to this collection of program units, and the binder places all of this into a single executable file.

There are four kinds of program units in Ada:

- subprograms (procedures and functions)
- packages
- generic units
- tasks

4.1. Subprograms (and Blocks)

A **subprogram** in Ada is a unit of sequential action, usually used to encapsulate the performance of a single logical "operation" with an optional set of parameters. A **block** in Ada is a subunit of sequential action found within a program unit like a subprogram. There are two kinds of subprograms: **procedures** (which return zero or more values) and **functions** (which return only one value). Blocks, procedures, and functions contain three parts:

- an optional declarative part, in which local variables are defined
- an executable statement part, in which code resides
- an optional exception handler, in which code to handle exceptional conditions (such as divide by zero) resides

The *declarative part* contains declarations of types and subtypes, variables and constants, and encapsulated program units (subprograms, packages, generic units, and tasks). The entities brought into existence in the declarative part only exist as long as the block, procedure, or function in which they reside is active.

The *executable statement part* contains executable statements, such as assignment and control statements.

The *exception handler* traps error conditions, or *exceptions*, and processes them.

Subprograms are the basic program units employed to perform sequential action in an Ada system. There are two classes of subprograms:

- **procedures** - accept and return values in parameter lists
- **functions** - accept values in parameter lists and only return one value

The optional parameter lists contain three classes of formal parameters:

- **in** - parameter values are passed into subprograms (internally, these parameters may only be used on the right-hand side of an assignment statement)
- **out** - parameter values are passed out of subprograms (internally, these parameters may only be used on the left-hand side of an assignment statement); **out** parameters may be used in procedures only
- **inout** - parameter values are passed both ways; **inout** parameters may be used in procedures only

Overloading Subprograms. Subprogram names may be *overloaded* (i.e., two or more subprograms may have the same name but different types or numbers of parameters), and Ada can resolve these from context.

Recursion in Subprograms. A subprogram may call itself, or *recurse*.

The Basic Differences Between Subprograms and Blocks. Subprograms can be compiled separately, while blocks are embedded in some larger unit which must be compiled as a whole. Embedded subprograms can only be placed in the declarative part of a program unit, while blocks can only be placed in the executable statement part. Subprograms can be invoked by a call, while blocks are invoked as part of the flow of execution only.

Blocks. The general form of a block is:

```
optional_label:
declare -- optional
  -- variable definitions
begin
  null;  -- statements
exception -- optional
  when others => null;  -- exception handler
end optional_label;
```

Example:

```
my_block:
declare
  I : INTEGER := 5;
begin
  Value := Value / I;  -- Value is an INTEGER external to the block
end my_block;
```

Subprograms: Functions. The general syntax of a function is:

```
function function_name ( parameters ) return type;
   -- specification
function function_name ( parameters ) return type is
   -- body
  -- optional variable definitions
begin
  -- statements (including at least one return statement)
exception -- optional
  when others => null;  -- exception handlers
end function_name;
```

Examples of function specifications and bodies:

```
function Sin (Angle : in FLOAT) return FLOAT:  -- spec
function Sin (Angle : in FLOAT) return FLOAT is -- body
begin
  null;  -- detail omitted
  return 1.0;  -- dummy return value
end Sin;

function Cos (Angle : FLOAT) return FLOAT;  -- mode is 'in' by default
function "*" (Left, Right : in COMPLEX_NUMBER) return COMPLEX_NUMBER;
  -- operator overloading
```

Examples of function calls:

```
X  := Sin (2.2);
```

```
X  := Cos (Angle => 45.2);
Y  := Trig_Lib.Cos (X);
C3 := Complex."*" (C1, C2);  -- prefix call
C3 := C1 * C2;  -- infix call, same as above prefix call
```

Subprograms: Procedures. The general syntax of a procedure is:

```
procedure procedure_name ( parameters );  -- specification
procedure procedure_name ( parameters) is  -- body
  -- optional local variables
begin
  -- statements (a return statement is not required)
exception -- optional
  when others => null;  -- exception handler
end procedure_name;
```

Examples of procedure specifications and bodies:

```
procedure Do_It;  -- specification

procedure Get_Status (Result : out STATUS);  -- spec
procedure Get_Status (Result : out STATUS) is -- body
begin
  Result := OK;
end Get_Status;

procedure Create (File : in out FILE_TYPE;
                  Name : in STRING := "DUMMY.TXT";
                  Mode : in FILE_MODE := IN_FILE);
   -- spec with default values for parameters
```

Examples of calls to the above procedures:

```
Do_It;
Get_Status (Value);
Get_Status (Result => Value);
Create (FD);  -- file name is "DUMMY.TXT" and mode is IN_FILE
Create (FD, Mode => OUT_FILE);  -- file name is still "DUMMY.TXT"
Create (File => FD,
        Name => "Myfile.txt",
        Mode => IN_FILE);
```

Problem 4.1: Write an Ada program which defines the type COMPLEX_NUMBER and functions "+", "-", "*", and PRINT which operate on objects of type COMPLEX_NUMBER. Illustrate the use of these functions as infix operators when possible.

4.2. Packages

A **package** is an encapsulation mechanism in Ada, allowing the programmer to collect groups of entities together. As a rule, these entities should be logically related. A **package** usually consists of two parts, like the other program units: a specification and a body.

Packages directly support object-oriented programming, providing a means to describe a class or object (an *abstract data type*). A package may implement either an object as an *abstract state machine* (wherein the state information is stored as data hidden in the body of the package) or a class (an *abstract data type*) using **private data types** (wherein the state information is stored as private data associated with each object).

Packages are used for four main purposes:

- as collections of constants and type declarations
- as collections of related functions
- as abstract state machines
- as abstract data types

Package bodies may contain an optional *initialization part*. If this is present, the code of the initialization part of a package is executed before the first line of code in the mainline procedure of an Ada system.

Packages may be embedded in blocks, subprograms, and any program unit in general.

A *private type* is a type declaration which is visible in the specification of a package, but its underlying implementation is hidden from the code withing the package and is of no concern to the outside world. *Private types* are the means of implementing *abstract data types* in Ada. In a package containing a *private type*, the only operations which may be performed on objects of that type are assignment, tests for equality and inequality, and the procedures and functions explicitly exported by the package. In a package containing a *limited private type*, the only operations which may be performed on objects of that type are the procedures and functions explicitly exported by the package.

The general form of a package specification is:

```
package package_name is
  type type_name is private;
    -- optional reference to a private type
  -- visible declarations
private
  -- private type declarations
end package_name;
```

The general form of a package body is:

```
package body package_name is
  -- implementations of code and hidden data
begin -- optional
  -- initialization statements
end package_name;
```

Examples of package specifications:

```
package Console is -- abstract state machine

  type LOCATION is record
    Row : INTEGER;
    Col : INTEGER;
  end record;

  procedure Clear_Screen;
  procedure Position_Cursor (Where : in LOCATION);
  procedure Write (Item : in CHARACTER);
  procedure Write (Item : in STRING);

end Console;
```

```
package Complex is -- abstract data type

  type OBJECT is private;

  function Set (Real : in FLOAT; Imag : in FLOAT) return OBJECT;
  function Real_Part (Item : in OBJECT);
  function Imag_Part (Item : in OBJECT);
  function "+" (Left, Right : in OBJECT) return OBJECT;
  function "-" (Left, Right : in OBJECT) return OBJECT;

private

  type OBJECT is record
    RP : FLOAT;
    IP : FLOAT;
  end record;

end Complex;
```

Problem 4.2: Write an Ada system containing a package COMPLEX which includes the functions written in problem 4.1. Set up package COMPLEX as an abstract data type.

4.3. Generic Units

Generic subprograms and packages, which are templates describing general-purpose algorithms that apply to a variety of types of data, may be created in Ada systems. The bodies of generic functions and procedures resemble normal subprograms except that the general types used in the specifications are employed rather than conventional types. These generic units describe the algorithms in general terms, specifying what kinds of restrictions must be placed on the data to which these templates apply. For example, certain algorithms require that the elements must have an associated test for equality and must be able to be assigned. Other algorithms require that the elements resemble integers or floating point numbers. The **generic formal parameters** presented in a specification of a generic unit explicitly identify the generic parameters and the restrictions placed upon them.

Generic functions look like this:

```
generic
  -- generic formal parameters
function function_name ( parameters ) return type;  -- spec
```

Generic procedure look like this:

```
generic
  -- generic formal parameters
procedure procedure_name ( parameters );  --spec
```

Generic packages look like this:

```
generic
  -- generic formal parameters
package package_name is  -- spec
  -- normal package stuff
end package_name;
```

An example of a full generic specification and body:

```
generic
  type ELEMENT is private;  -- anything that can be assigned
procedure Exchange (Item1, Item2 : in out ELEMENT);  -- spec

procedure Exchange (Item1, Item2 : in out ELEMENT) is  -- body
  Temp : ELEMENT;  -- temporary is the same type as the parameters
begin
  Temp := Item1;
  Item1 := Item2;
  Item2 := Temp;
end Exchange;
```

To make use of a generic unit, it must be instantiated. Any generic parameters must be specified during the instantiation, specifying what external, existing entities are to corresponding to those generic parameters. An example:

```
type X is record
  I : INTEGER;
  F : FLOAT;
end record;
procedure Int_Exchange is new Exchange (ELEMENT => INTEGER);
procedure Float_Exchange is new Exchange (FLOAT);
procedure X_Exchange is new Exchange (X);

I1, I2 : INTEGER;
F1, F2 : FLOAT;
X1, X2 : X;

Int_Exchange (I1, I2);  -- actually use the new procedures
Float_Exchange (F1, F2);
X_Exchange (X1, X2);
```

Hence, the generic formal parameters are the key to understanding, writing, and using generic units. There are three kinds of generic formal parameters: types, objects, and subprograms.

l Types as generic formal parameters:

Type Parameter	*Operations Allowed*	*Applicable Data Types*
type T is private;	= /= :=	All assignable
type T is limited private;	- none -	All
type D is (<>);	= /= := > >= < <=	Discrete
	PRED SUCC FIRST LAST	

type I is range <>;	Integer operations	Integer
type F is digits <>;	Real operations	Float
type FIXED is delta <>;	Fixed point operations	Fixed

- Object declarations may appear as formal parameters.
- Subprograms may appear as formal parameters.

Problem 4.3: Write an Ada program which includes a generic procedure which implements a bubble sort. Instantiate this to sort arrays of integers, floats, and records. Display the arrays before and after the sort.

4.4. Tasks

In Ada, one can write programs that perform more than one activity concurrently. This concurrent processing is called *tasking*, and the units of code that run concurrently are called **tasks**.

A simple format for task specifications and bodies:

```
task task_name;  -- specification
task body task_name is -- body
  -- local variable declarations
begin
  -- code
end task_name;
```

A more complex format:

```
task task_name is -- specification
  entry entry_name ( parameters );
end task_name;

task body task_name is  -- body
  -- local variables
begin
  accept entry_name ( parameters ) do -- code follows
    -- code at entry point
  end entry_name;
end task_name;
```

The entry statement in the task specification identifies the entry points to the task. The accept statement in the task body identifies the code to be executed at that entry point. The entry points to a task are called like subprogram calls from other program units.

A task type may be created as well:

```
task type task_name1;  -- specification
task body task_name1 is -- body
  -- local variable declarations
begin
  -- code
end task_name1;

task type task_name2 is -- specification
  entry entry_name ( parameters );
end task_name2;

task body task_name2 is  -- body
  -- local variables
begin
  accept entry_name ( parameters ) do -- code follows
    -- code at entry point
  end entry_name;
end task_name2;
```

When we have task types, we have the ability to statically or dynamically create task objects:

```
my_task : task_name1;  -- task is running right after definition
task_vector : array (1..20) of task_name2;  -- array of tasks
```

The interfacing of two tasks in order to pass data is called a *rendezvous* in Ada. Entry points are called

like subprograms to effect a rendezvous.

Problem 4.4: Write an Ada system that consists of a task type that accepts a name string and an amount for the task to delay, delays the indicated amount, and displays a message when done. In your mainline, create and start an array of 5 tasks of this type.

4.5. Exceptions

Two kinds of errors are commonly encountered in programming: compilation errors and runtime errors. In Ada, runtime errors are called *exceptions*. Exceptions may be predefined or user-defined. To define an exception:

```
Exception_name : exception;
```

To raise an exception to be handled by an exception handler somewhere in the calling chain:

```
raise Exception_name;
```

Exception handlers are Ada constructs that handle exceptions. An exception handler is placed at the end of a block, subprogram, package, or task, and is denoted by the keyword exception followed by the text of the handler code. Example (for a block):

```
begin -- note that I is defined external to this block
  I := I / 0;  -- division by zero
exception
  when NUMERIC_ERROR =>
    I := 10;
end;
```

Predefined Exceptions. There are two main sources for predefined exceptions in Ada: package STANDARD and package TEXT_IO:

- Package STANDARD
  - CONSTRAINT_ERROR - raised whenever a value goes out of bounds, such as assigning a value of 11 to a variable whose type is in the range from 0 to 10
  - NUMERIC_ERROR - raised when illegal or unmanageable mathematical operations are performed, such as dividing by zero
  - STORAGE_ERROR - raised when the computer runs out of available memory
  - TASKING_ERROR - raised when there is a problem with the multitasking environment, such as calling a task which is no longer active
  - PROGRAM_ERROR - all other exceptions not covered by the above or other exceptions defined by the user or some other package, such as reaching the end of a function without hitting a return statement
- Package TEXT_IO
  - DATA_ERROR - encountered an input that is not of the required type
  - DEVICE_ERROR - malfunction in the underlying system, such as disk space being full
  - END_ERROR - an attempt was made to read past the end of the file
  - LAYOUT_ERROR - raised by COL, LINE, or PAGE if the value returned exceeds COUNT'LAST
  - MODE_ERROR - an attempt was made to read from or test the end of a file whose current mode is OUT_FILE, or an attempt was made to write to a file whose current mode is IN_FILE
  - NAME_ERROR - the string given as a file name to CREATE or OPEN does not allow the identification of an external file
  - STATUS_ERROR - an attempt was made to operate on a file that is not open or open a file that is already open

- USE_ERROR - an operation is attempted that is not possible for reasons that depend on the characteristics of the external file

Exception Propagation. If the program unit that raises an exception does not contain an exception handler that handles the exception, the exception is propagated to the next level beyond the unit. This level varies, depending on the unit raising the exception:

- If the unit is a mainline procedure, the Ada runtime environment handles the exception by aborting the program.

- If the unit is a block, the exception is passed to the program unit (or block) containing the block that raised the exception.

- If the unit is a subprogram, the exception is passed to the program unit or block that called the subprogram.

The propagation path of an exception is determined at runtime.

To reraise the current exception in an exception handler, the statement

```
raise;
```

may be used.

Suppressing Exceptions. Ada performs many checks at runtime to ensure that array indices are not exceeded, variables stay within range, etc. If these checks fail, *exceptions* are raised. This results in larger code and slower execution speed in general.

In certain real-time applications, where space and time constraints are critical, runtime error checking may be too expensive. A solution is to use *exception suppression*.

Exception suppression turns off runtime error checking. It is implemented by a *pragma* (a compiler directive) called SUPPRESS:

```
pragma SUPPRESS (RANGE_CHECK);
  -- turns off range checking on array indices and variable values
pragma SUPPRESS (RANGE_CHECK, INTEGER);
  -- turns off range checking on integers only
pragma SUPPRESS (RANGE_CHECK, X);
  -- turns off range checking for a particular object
```

Problem 4.5: Write an Ada system that contains its own exception, DIVIDE_BY_ZERO. Write a function "/" that divides two integers, checking first to see if the second integer is zero (in which case it raises DIVIDE_BY_ZERO). Your mainline enter a loop, dividing several integer pairs and printing the result. If the exception DIVIDE_BY_ZERO is raised, the mainline should print out that this happened, show the attempted division, and continue with the rest of the integer pairs.

A. Suggested Reading

David J. Naiditch, **Rendezvous with Ada: A Programmer's Introduction**, John Wiley & Sons, 1989, ISBN 0-471-61654-0

Grady Booch, **Software Engineering with Ada, 2nd Edition**, Benjamin/Cummings Publishing Company, 1987, ISBN 0-8053-0604-8

Grady Booch, **Software Components with Ada**, Benjamin/Cummings Publishing Company, 1987, ISBN 0-8053-0610-2 (this is a more advanced text - not for the beginner)

Michael B. Feldman and Elliot B. Koffman, **Ada Problem Solving and Program Design**, Addison-Wesley Publishing Company, 1992, ISBN 0-201-50006-X

Karl A. Nyberg (editor), **The Annotated Ada Reference Manual, 2nd Edition**, Grebyn Corporation, 1991, ISBN unknown (this is a reference manual - not a tutorial)

B. Specification of Package Text_IO

```
µwith io_exceptions;
package text_io is

   type file_type is limited private;

   type file_mode is (in_file, out_file);

   type count             is range 0 .. integer'last;
   subtype positive_count  is count range 1 .. count'last;
   unbounded: constant count := 0;    -- line and page length

   subtype field          is integer range 0 .. integer'last;
   subtype number_base     is integer range 2 .. 16;

   type type_set is (lower_case, upper_case);

   -- file management

   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;

   -- control of default input and output files

   procedure set_input    (file:    in file_type);
   procedure set_output   (file:    in file_type);

   function standard_input     return file_type;
   function standard_output    return file_type;
   function standard_error     return file_type;

   function current_input      return file_type;
   function current_output     return file_type;

   -- 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
```

```
procedure new_line  (file:       in file_type;
                     spacing:    in positive_count := 1);
procedure new_line  (spacing:    in positive_count := 1);

procedure skip_line (file:       in file_type;
                     spacing:    in positive_count := 1);
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);
procedure new_page;

procedure skip_page  (file:    in file_type);
procedure skip_page;

function end_of_page (file:    in file_type) return boolean;
function end_of_page return boolean;

function end_of_file (file:    in file_type) return boolean;
function end_of_file return boolean;

procedure set_col    (file:    in file_type; to:    in positive_count);
procedure set_col    (to:      in positive_count);

procedure set_line   (file:    in file_type; to:    in positive_count);
procedure set_line   (to:      in positive_count);

function col         (file:    in file_type) return positive_count;
function col return positive_count;

function line        (file:    in file_type) return positive_count;
function line return positive_count;

function page        (file:    in file_type) return positive_count;
function page return positive_count;

-- character input-output

procedure get    (file:    in file_type; item:    out character);
procedure get    (item:    out character);
procedure put    (file:    in file_type; item:    in character);
procedure put    (item:    in character);

-- string input-output

procedure get    (file:    in file_type; item:    out string);
procedure get    (item:    out string);
procedure put    (file:    in file_type; item:    in string);
procedure put    (item:    in string);

procedure get_line (file: in file_type;
                    item: out string; last: out natural);
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 package for input-output of integer types

generic
   type num is range <>;
package integer_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;
                    base:    in number_base := default_base);

end integer_io;

-- generic package for input-output of real types

generic
   type num is digits <>;
package float_io is
   default_fore:   field := 2;
   default_aft:    field := num'digits - 1;
   default_exp:    field := 3;


   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);

   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 float_io;

generic
   type num is delta <>;
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);

    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 enum is (<>);
  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);

    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;

  -- 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;

private
   type file_object;
   type file_type is access file_object;
   -- detail omitted
end text_io;
```

§

C. Solutions to Problems

Solution 1.1:

Code

```
µ-- Problem 1.1
-- by Rick Conn
with Text_IO;  -- context specification

procedure Count_Down is  -- an Ada mainline is always a procedure

  Number_of_Iterations : Integer := 1;
    -- local variable definition

  package Int_IO is new Text_IO.Integer_IO (Integer);
    -- local package for Integer I/O

begin -- Count_Down

  Text_IO.Put ("Enter number of iterations: ");
  Int_IO.Get (Number_of_Iterations);
  Text_IO.New_Line;

  Text_IO.Put ("The loop will run for ");
  Int_IO.Put (Number_of_Iterations);
  Text_IO.Put_Line (" iterations");

  for Index in 1 .. Number_of_Iterations loop
    Text_IO.Put ("Iteration: ");
    Int_IO.Put (Index, 4);  -- 4 column field
    Text_IO.New_Line;
  end loop;

end Count_Down;
```

```
§
```

Output

```
µEnter number of iterations: 12
The loop will run for          12 iterations
Iteration:    1
Iteration:    2
```

```
Iteration:  3
Iteration:  4
Iteration:  5
Iteration:  6
Iteration:  7
Iteration:  8
Iteration:  9
Iteration: 10
Iteration: 11
Iteration: 12
```

§

Solution 1.2:

Code

```
µ-- Problem 1.2
-- by Rick Conn
with Text_IO;  -- context clause

procedure Display is

  -- display a single quote and a double quote

begin

  Text_IO.Put_Line ("First Method --");

  Text_IO.Put_Line (" Single Quote: '");
  Text_IO.Put_Line (" Double Quote: """);

  Text_IO.Put_Line ("Alternate Method --");

  Text_IO.Put (" Single Quote: "); Text_IO.Put ('');  Text_IO.New_Line;
  Text_IO.Put (" Double Quote: "); Text_IO.Put ('"');  Text_IO.New_Line;

end Display;
```

§

Output

```
µFirst Method --
 Single Quote: '
 Double Quote: "
Alternate Method --
 Single Quote: '
 Double Quote: "
```

§

Solution 1.3:

Code

```
µ-- Problem 1.3
-- by Rick Conn
package example is

  function square (item : in float) return float;  -- spec

  procedure display (item : in integer);  -- spec

  procedure display (item : in float);  -- spec
    -- overloading of the procedure display with different
    -- parameter types is fine in Ada

end example;

with text_io;
package body example is

  package int_io is new text_io.integer_io (integer);
  package flt_io is new text_io.float_io (float);

  function square (item : in float) return float is -- body
  begin
    return item * item;
  end square;

  procedure display (item : in integer) is  -- body
  begin
    text_io.put ("Integer number: ");
    int_io.put (item, 4);
    text_io.new_line;
  end display;

  procedure display (item : in float) is  -- body
  begin
    text_io.put ("Floating point number: ");
    flt_io.put (item, 5, 4, 0);
    text_io.new_line;
  end display;

end example;

with example;
```

```
procedure demo is

  result : float;

begin

  result := example.square (2.2);
  example.display (result);  -- float display
  example.display (2);       -- integer display

end demo;
```

§

Output

```
µFloating point number:     4.8400
Integer number:    2
```

§

Solution 1.4:

Code

```
µ-- Problem 1.4
-- by Rick Conn
with text_io;
procedure Display_ASCII is
  package Int_IO is new Text_IO.Integer_IO (Integer);

begin -- Display_ASCII

  for I in ASCII.NUL .. ASCII.SUB loop

    Text_IO.Put ("Value: ");
    Int_IO.Put (Character'Pos(I), 3);
      -- the attribute 'Pos converts a character into
      -- its position number (value) in the ASCII
      -- character set
    Text_IO.New_Line;

  end loop;

end Display_ASCII;
```

```

```

§

Output

```
µValue:   0
Value:   1
Value:   2
Value:   3
Value:   4
Value:   5
Value:   6
Value:   7
Value:   8
Value:   9
Value:  10
Value:  11
Value:  12
Value:  13
Value:  14
Value:  15
Value:  16
Value:  17
Value:  18
Value:  19
Value:  20
Value:  21
Value:  22
Value:  23
Value:  24
Value:  25
Value:  26
```

§

Solution 1.5:

Code

```
µ-- Problem 1.5
-- by Rick Conn
with Text_IO;
procedure Reserved_Demo is

  package Int_IO is new Text_IO.Integer_IO (Integer);

begin

  for I in 1 .. 15 loop

    if I < 6 then
      Int_IO.Put (I, 2);
    elsif I = 6 then
      Int_IO.Put (I, 2);
      Text_IO.New_Line;
    elsif I < 10 then
      Int_IO.Put (I, 2);
    elsif I = 10 then
      Int_IO.Put (I, 3);
      Text_IO.New_Line;
    else
      Text_IO.Put ("Too Big: ");
      Int_IO.Put (I, 3);
      Text_IO.New_Line;
    end if;

  end loop;

end Reserved_Demo;
```

```
§
```

Output

```
µ 1 2 3 4 5 6
 7 8 9 10
Too Big:  11
Too Big:  12
Too Big:  13
Too Big:  14
Too Big:  15
```

§

Solution 1.6:

Code

```
µ-- Problem 1.6
-- by Rick Conn
with Text_IO;
procedure Truth_Table is

  package Bool_IO is new Text_IO.Enumeration_IO (Boolean);

begin -- Truth_Table

  Text_IO.Put_Line ("   I     J      I and J  I or  J  I xor J");
  for I in FALSE .. TRUE loop

    for J in FALSE .. TRUE loop

      Bool_IO.Put (I, 6);
      Bool_IO.Put (J, 6);
      Text_IO.Put ("    ");
      Bool_IO.Put (I and J, 7);
      Text_IO.Put ("  ");
      Bool_IO.Put (I or J, 7);
      Text_IO.Put ("  ");
      Bool_IO.Put (I xor J, 7);
      Text_IO.New_Line;

    end loop;

  end loop;

end Truth_Table;
```

§

Output

```
µ   I     J      I and J  I or  J  I xor J
FALSE FALSE     FALSE    FALSE    FALSE
FALSE TRUE      FALSE    TRUE     TRUE
TRUE  FALSE     FALSE    TRUE     TRUE
TRUE  TRUE      TRUE     TRUE     FALSE
```

§

Solution 2.1:

Code

```
µ-- Problem 2.1
-- by Rick Conn
package Colors is

  type COLOR is (RED, GREEN, BLUE, YELLOW, ORANGE,
                 TURQUOISE, MAGENTA, VIOLET);

  procedure Display (Last_Color : in COLOR);

end Colors;

with Text_IO;  -- context clause
package body Colors is

  procedure Display (Last_Color : in COLOR) is
  begin
    Text_IO.Put_Line ("Color Display:");
    for C in COLOR'FIRST .. Last_Color loop
      Text_IO.Put_Line ("   " & COLOR'IMAGE(C));
        -- string concatenation is used to offset the image
    end loop;
  end Display;

end Colors;

with Colors;
procedure Main is
begin
  Colors.Display(Colors.ORANGE);
  Colors.Display(Colors.RED);
  Colors.Display(Colors.VIOLET);
end Main;
```

§

Output

```
µColor Display:
   RED
   GREEN
   BLUE
   YELLOW
```

```
   ORANGE
Color Display:
   RED
Color Display:
   RED
   GREEN
   BLUE
   YELLOW
   ORANGE
   TURQUOISE
   MAGENTA
   VIOLET
```

§

Solution 2.2:

Code

```
µ-- Problem 2.2
-- by Rick Conn
with Text_IO;
procedure Main is

  type SPEED is new FLOAT range 0.0 .. 1_000_000.0;      -- MPH
  type TIME is new FLOAT range 0.0 .. 24.0;              -- hours
  type DISTANCE is new FLOAT range 0.0 .. 24_000_000.0;  -- miles

  Current_Speed     : SPEED;
  Elapsed_Time      : TIME;
  Computed_Distance : DISTANCE;

  package Float_IO is new Text_IO.Float_IO (FLOAT);

  function "*" (S : in SPEED;
                T : in TIME)
     return DISTANCE is
  begin
    return DISTANCE (FLOAT(S) * FLOAT(T));
  end "*";

begin -- Main

  Text_IO.Put_Line ("Speed      Time  Distance");
  for S in 0 .. 5 loop
    Current_Speed := SPEED (FLOAT(S) * 200_000.0);

    for T in 0 .. 3 loop
      Elapsed_Time := TIME (FLOAT(T) * 8.0);

      Computed_Distance := Current_Speed * Elapsed_Time;

      Float_IO.Put (FLOAT(Current_Speed), 7, 1, 0);
      Text_IO.Put("  ");
      Float_IO.Put (FLOAT(Elapsed_Time), 2, 1, 0);
      Text_IO.Put("  ");
      Float_IO.Put (FLOAT(Computed_Distance), 8, 1, 0);
      Text_IO.New_Line;

    end loop;
  end loop;
```

```
end Main;
```

§

Output

```
µSpeed      Time  Distance
     0.0   0.0         0.0
     0.0   8.0         0.0
     0.0  16.0         0.0
     0.0  24.0         0.0
 200000.0   0.0         0.0
 200000.0   8.0   1600000.0
 200000.0  16.0   3200000.0
 200000.0  24.0   4800000.0
 400000.0   0.0         0.0
 400000.0   8.0   3200000.0
 400000.0  16.0   6400000.0
 400000.0  24.0   9600000.0
 600000.0   0.0         0.0
 600000.0   8.0   4800000.0
 600000.0  16.0   9600000.0
 600000.0  24.0  14400000.0
 800000.0   0.0         0.0
 800000.0   8.0   6400000.0
 800000.0  16.0  12800000.0
 800000.0  24.0  19200000.0
1000000.0   0.0         0.0
1000000.0   8.0   8000000.0
1000000.0  16.0  16000000.0
1000000.0  24.0  24000000.0
```

§

Solution 2.3:

Code

```
µ-- Problem 2.3
-- by Rick Conn
with Text_IO;
procedure Main is

  type SQUARE is (EMPTY, CROSS, ZERO);
  type ROW is (ROW_1, ROW_2, ROW_3);
  type COL is (COL_1, COL_2, COL_3);
  type TIC_TAC_TOE_BOARD is array (ROW, COL) of SQUARE;

  TTT_1 : TIC_TAC_TOE_BOARD := (ROW_1 => (EMPTY, CROSS, CROSS),
                        ROW_2 => (ZERO, EMPTY, EMPTY),
                        ROW_3 => (ZERO, ZERO, CROSS));

  TTT_2 : TIC_TAC_TOE_BOARD := (ROW_1 => (CROSS, ZERO, CROSS),
                        ROW_2 => (ZERO, ZERO, EMPTY),
                        ROW_3 => (ZERO, CROSS, CROSS));

  procedure Display (TTT : in TIC_TAC_TOE_BOARD) is
    package Square_IO is new Text_IO.Enumeration_IO (SQUARE);
  begin -- Display
    Text_IO.Put_Line ("TTT Board:");
    for R in ROW loop
      Text_IO.Put ("  ");
      for C in COL loop
        Square_IO.Put (TTT_1 (R, C), 6);
      end loop;
      Text_IO.New_Line;
    end loop;
  end Display;

begin -- Main

  Display (TTT_1);
  Display (TTT_2);

end Main;
```

§

Output

```
µTTT Board:
  EMPTY CROSS CROSS
  ZERO  EMPTY EMPTY
  ZERO  ZERO  CROSS
TTT Board:
  EMPTY CROSS CROSS
  ZERO  EMPTY EMPTY
  ZERO  ZERO  CROSS
```

§

Solution 2.4:

Code

```
µ-- Problem 2.4
-- by Rick Conn
with Text_IO;
procedure Main is

  type String_Rec is record
    S : STRING(1..80);
    L : NATURAL;  -- last index
  end record;

  SRA : array (1..5) of String_Rec;

begin -- Main

  Text_IO.Put_Line ("Enter 5 strings:");

  for I in 1 .. 5 loop
    Text_IO.Put ("String " & NATURAL'IMAGE(I) & ": ");
    Text_IO.Get_Line (SRA(I).S, SRA(I).L);
  end loop;

  for I in 1 .. 5 loop
    Text_IO.Put ("Output String " & NATURAL'IMAGE(I) & ": ");
    Text_IO.Put_Line (SRA(I).S (1 .. SRA(I).L));
      -- output a slice of the string
  end loop;

end Main;
```

```
§
```

Output

```
µEnter 5 strings:
String  1: this is a test
String  2: this is only a test
String  3: this is fun
String  4: I love Ada
String  5: bye for now
Output String  1: this is a test
Output String  2: this is only a test
Output String  3: this is fun
```

```
Output String  4: I love Ada
Output String  5: bye for now
```

§

Solution 2.5:

Code

```
µ-- Problem 2.5
-- by Rick Conn
with Text_IO;
procedure Main is

  type BOOLEAN_VECTOR is array (1..4) of BOOLEAN;
  T : constant BOOLEAN := TRUE;
  F : constant BOOLEAN := FALSE;

  A : constant BOOLEAN_VECTOR := (F, F, T, T);
  B : constant BOOLEAN_VECTOR := (F, T, F, T);

  B_AND : BOOLEAN_VECTOR;
  B_OR  : BOOLEAN_VECTOR;
  B_XOR : BOOLEAN_VECTOR;

  package Bool_IO is new Text_IO.Enumeration_IO (BOOLEAN);

begin -- Main

  B_AND := A and B;
  B_OR  := A or B;
  B_XOR := A xor B;

  Text_IO.Put_Line ("A     B     and   or    xor");
  for I in 1 .. 4 loop
    Bool_IO.Put (A(I), 6);
    Bool_IO.Put (B(I), 6);
    Bool_IO.Put (B_AND(I), 6);
    Bool_IO.Put (B_OR(I), 6);
    Bool_IO.Put (B_XOR(I), 6);
    Text_IO.New_Line;
  end loop;

end Main;
```

§

Output

```
µA    B     and   or    xor
FALSE FALSE FALSE FALSE FALSE
```

```
FALSE TRUE  FALSE TRUE  TRUE
TRUE  FALSE FALSE TRUE  TRUE
TRUE  TRUE  TRUE  TRUE  FALSE
```

§

Solution 2.6:

Code

```
µ-- Problem 2.6
-- by Rick Conn
with Text_IO;
procedure Main is

  type FARRAY is array (POSITIVE range <>) of FLOAT;

  FA1 : FARRAY(1..5) := (2.2, 3.3, 4.4, 5.5, 7.2);
  FA2 : constant FARRAY := (12.2, 10.2, 1.0, 5.2);
  FA3 : constant FARRAY := (1.1, 2.2);

  package Float_IO is new Text_IO.Float_IO (FLOAT);

  procedure Display_FARRAY (Item : in FARRAY) is
  begin
    for I in Item'RANGE loop
      Text_IO.Put ("  ");
      Float_IO.Put (Item(I), 2, 1, 0);
    end loop;
    Text_IO.New_Line;
  end Display_FARRAY;

begin -- Main

  Display_FARRAY (FA1);
  Display_FARRAY (FA2);
  Display_FARRAY (FA3);

end Main;

§
```

Output

```
µ   2.2   3.3   4.4   5.5   7.2
  12.2  10.2   1.0   5.2
   1.1   2.2

§
```

Solution 2.7:

Code

```
µ-- Problem 2.7
-- by Rick Conn
with Text_IO;
procedure Main is

 type BOOK is record
  Title      : STRING(1..80);
  Title_Last  : NATURAL;
  Author     : STRING(1..80);
  Author_Last : NATURAL;
  Price      : FLOAT;
  Year       : INTEGER;
 end record;

 package Float_IO is new Text_IO.Float_IO (FLOAT);
 package Int_IO is new Text_IO.Integer_IO (INTEGER);

 function Setup (Title  : in STRING;
                 Author : in STRING;
                 Price  : in FLOAT;
                 Year   : in INTEGER) return BOOK is
  Local_Book : BOOK;
 begin
  Local_Book.Title(1..Title'LENGTH) := Title;
  Local_Book.Title_Last := Title'LENGTH;
  Local_Book.Author(1..Author'LENGTH) := Author;
  Local_Book.Author_Last := Author'LENGTH;
  Local_Book.Price := Price;
  Local_Book.Year  := Year;
  return Local_Book;
 end Setup;

 procedure Display (Item : in BOOK) is
 begin
  Text_IO.Put_Line ("Title: " & Item.Title(1..Item.Title_Last));
  Text_IO.Put_Line ("  Author: " & Item.Author(1..Item.Author_Last));
  Text_IO.Put ("  Price: $");
    Float_IO.Put (Item.Price, 2, 2, 0);
    Text_IO.New_Line;
  Text_IO.Put ("  Year: ");
    Int_IO.Put (Item.Year, 4);
    Text_IO.New_Line;
```

```
  end Display;

begin -- Main

  declare
    My_Book   : BOOK;
    His_Book  : BOOK;
    Your_Book : BOOK;
  begin  -- block
    My_Book   := Setup ("War and Peace",
                        "Tolstoy",
                        39.95,
                        1912);
    His_Book  := Setup ("Software Engineering with Ada",
                        "Booch",
                        19.95,
                        1987);
    Your_Book := Setup ("Fun and Games",
                        "Dick and Jane",
                        5.95,
                        1990);

    Display (My_Book);
    Display (His_Book);
    Display (Your_Book);
  end;

end Main;
```

§

Output

```
µTitle: War and Peace
  Author: Tolstoy
  Price: $39.95
  Year: 1912
Title: Software Engineering with Ada
  Author: Booch
  Price: $19.95
  Year: 1987
Title: Fun and Games
  Author: Dick and Jane
  Price: $ 5.95
  Year: 1990
```

§

Solution 2.8:

Code

```
µ-- Problem 2.8
-- by Rick Conn
with Text_IO;
procedure Main is

  package Float_IO is new Text_IO.Float_IO (FLOAT);

  type FARRAY is array (1..100) of FLOAT;

  type REC is record
    Count  : NATURAL;
    Vector : FARRAY;
  end record;
  type REC_PTR is access REC;

  P, P2 : REC_PTR;

  procedure Display (Ptr : in REC_PTR) is
  begin
    for I in 1 .. Ptr.Count loop
      Float_IO.Put (Ptr.Vector(I), 5, 5, 0);
    end loop;
    Text_IO.New_Line;
  end Display;

begin -- Main

  P := new REC;
  P2 := new REC;
  P.Count := 4;
  P.Vector(1..4) := (2.2, 1.1, 3.3, 4.4);
  P2.Count := 2;
  P2.Vector(1..2) := (10.2, 12.2);
  Display (P);
  Display (P2);

end Main;
```

```
§
```

Output

```
μ    2.20000    1.10000    3.30000    4.40000
  10.20000   12.20000
```

```
§
```

Solution 2.9:

Code

```
µ-- Problem 2.9
-- by Rick Conn
with Text_IO;
procedure Main is

  type BIT is (OFF, ON);
  for BIT'SIZE use 1;
  for BIT use (OFF => 0, ON => 1);

  type BINIT is array (1..2) of BIT;
  pragma Pack (BINIT);

  type STATUS_WORD is record
    Carry_Bit    : BIT;
    Overflow_Bit : BIT;
    Zero_Bit     : BIT;
    Special      : BINIT;
  end record;
  for STATUS_WORD'SIZE use 5;
  for STATUS_WORD use record
    Carry_Bit    at 0 range 0..0;
    Overflow_Bit at 0 range 1..1;
    Zero_Bit     at 0 range 2..2;
    Special      at 0 range 3..4;
  end record;

  One_Status : STATUS_WORD := (Carry_Bit    => ON,
                               Overflow_Bit => OFF,
                               Zero_Bit     => ON,
                               Special      => (OFF, ON));

  Two_Status : STATUS_WORD := (Carry_Bit    => OFF,
                               Overflow_Bit => OFF,
                               Zero_Bit     => OFF,
                               Special      => (OFF, OFF));

  Tre_Status : STATUS_WORD := (Carry_Bit    => ON,
                               Overflow_Bit => ON,
                               Zero_Bit     => ON,
                               Special      => (ON, ON));

  procedure Display (Item : in STATUS_WORD) is
```

```
  begin
    Text_IO.Put_Line ("Carry_Bit      : " &
       BIT'IMAGE(Item.Carry_Bit));
    Text_IO.Put_Line ("  Overflow_Bit : " &
       BIT'IMAGE(Item.Overflow_Bit));
    Text_IO.Put_Line ("  Zero_Bit     : " &
       BIT'IMAGE(Item.Zero_Bit));
    Text_IO.Put_Line ("  Special      : " &
       BIT'IMAGE(Item.Special(1)) &
       BIT'IMAGE(Item.Special(2)));
  end Display;

begin -- Main

  Display (One_Status);
  Display (Two_Status);
  Display (Tre_Status);

end Main;
```

§

Output

```
µCarry_Bit      : ON
  Overflow_Bit : OFF
  Zero_Bit     : ON
  Special      : OFFON
Carry_Bit      : OFF
  Overflow_Bit : OFF
  Zero_Bit     : OFF
  Special      : OFFOFF
Carry_Bit      : ON
  Overflow_Bit : ON
  Zero_Bit     : ON
  Special      : ONON
```

§

Solution 2.10:

Code

```
µ-- Problem 2.10
-- by Rick Conn
with Text_IO;
procedure Main is

  I, J : INTEGER;

  package Int_IO is new Text_IO.Integer_IO (INTEGER);

begin -- Main

  loop
    Text_IO.Put ("Enter 2 numbers (0 0 to quit): ");
    Int_IO.Get (I);  Int_IO.Get(J);
    Text_IO.New_Line;
    exit when I = 0 and J = 0;
    if J /= 0 and then I/J > 0 then
      Int_IO.Put (I/J);
    else
      Text_IO.Put ("Zero");
    end if;
    Text_IO.New_Line;
  end loop;

end Main;
```

```
§
```

Output

```
µEnter 2 numbers (0 0 to quit): 12 2

         6
Enter 2 numbers (0 0 to quit): 5 3

         1
Enter 2 numbers (0 0 to quit): 1 2

Zero
Enter 2 numbers (0 0 to quit): 0 0
```

§

Solution 3.1:

Code

```
µ-- Problem 3.1
-- by Rick Conn
with Text_IO;
procedure Main is

  package Int_IO is new Text_IO.Integer_IO (INTEGER);

begin

  declare
    Sum : INTEGER := 0;

    procedure Display (Message : in STRING; Value : in INTEGER) is
    begin
      Text_IO.Put (Message & ": ");
      Int_IO.Put (Value, 5);
      Text_IO.New_Line;
    end Display;

  begin -- block

    for I in 1 .. 10 loop

      Sum := Sum + I;
      Display ("The sum so far is", Sum);

    end loop;

  end;  -- block

end Main;
```

```
§
```

Output

```
µThe sum so far is:     1
The sum so far is:     3
The sum so far is:     6
The sum so far is:    10
The sum so far is:    15
The sum so far is:    21
```

```
The sum so far is:    28
The sum so far is:    36
The sum so far is:    45
The sum so far is:    55
```

§

Solution 3.2:

Code

```
µ-- Problem 3.2
-- by Rick Conn
with Text_IO;
procedure Main is

  type CHAR_TYPE is (LOWER_CASE, UPPER_CASE, SHARP, COMMA,
                     SEMICOLON, DIGIT, SOME_OTHER);

  Char_List : constant STRING := "a#B,;2c8&1";

  package Char_Type_IO is new Text_IO.Enumeration_IO (CHAR_TYPE);

  function Char_Class (Item : in CHARACTER) return CHAR_TYPE is
    Result : CHAR_TYPE;
  begin
    case Item is
      when 'A' .. 'Z' => Result := UPPER_CASE;
      when 'a' .. 'z' => Result := LOWER_CASE;
      when '0' .. '9' => Result := DIGIT;
      when '#'        => Result := SHARP;
      when ','        => Result := COMMA;
      when ';'        => Result := SEMICOLON;
      when others     => Result := SOME_OTHER;
    end case;
    return Result;
  end Char_Class;

begin -- Main

  for I in Char_List'RANGE loop
    Text_IO.Put ("Character " & Char_List(I) & " is of type ");
    Char_Type_IO.Put (Char_Class(Char_List(I)));
    Text_IO.New_Line;
  end loop;

end Main;
```

§

Output

```
µCharacter a is of type LOWER_CASE
```

```
Character # is of type SHARP
Character B is of type UPPER_CASE
Character , is of type COMMA
Character ; is of type SEMICOLON
Character 2 is of type DIGIT
Character c is of type LOWER_CASE
Character 8 is of type DIGIT
Character & is of type SOME_OTHER
Character 1 is of type DIGIT
```

§

Solution 3.3:

Code

```
µ-- Problem 3.3
-- by Rick Conn
with Text_IO;
procedure Main is

  Inline : STRING (1..80);
  Inlast : NATURAL;

begin

  for I in 1 .. 10 loop

    Text_IO.Put ("Enter String: ");
    Text_IO.Get_Line (Inline, Inlast);

    if Inlast >= 4 and then Inline(1..4) = "EXIT" then
      Text_IO.Put_Line ("Leaving Loop on EXIT Command");
      exit;
    end if;

  end loop;

end Main;
```

§

Output

```
µFirst run --

Enter String: this is a teest
Enter String: it is fun
Enter String: to write programs
Enter String: in Ada
Enter String: exit
Enter String: EXIT now
Leaving Loop on EXIT Command

Second run --

Enter String: a
Enter String: b
```

```
Enter String: c
Enter String: d
Enter String: e
Enter String: f
Enter String: g
Enter String: h
Enter String: i
Enter String: j
```

§

Solution 4.1:

Code

```
µ-- Problem 4.1
-- by Rick Conn
with Text_IO;
procedure Main is

  type COMPLEX_NUMBER is record
    RP : FLOAT := 0.0;  -- initialize to 0,0
    IP : FLOAT := 0.0;
  end record;

  N1, N2, N3 : COMPLEX_NUMBER;

  package Float_IO is new Text_IO.Float_IO (FLOAT);

  function "+" (Left, Right : in COMPLEX_NUMBER)
     return COMPLEX_NUMBER is
  begin
    return (Left.RP + Right.RP, Left.IP + Right.IP);
  end "+";

  function "-" (Left, Right : in COMPLEX_NUMBER)
     return COMPLEX_NUMBER is
  begin
    return (Left.RP - Right.RP, Left.IP - Right.IP);
  end "-";

  function "*" (Left, Right : in COMPLEX_NUMBER)
     return COMPLEX_NUMBER is
  begin
    return (Left.RP * Left.RP - Right.RP * Right.RP,
          Left.RP * Right.IP + Left.IP * Right.RP);
  end "*";

  procedure Print (Item : in COMPLEX_NUMBER) is
  begin
    Float_IO.Put (Item.RP, 5, 5, 0);
    Text_IO.Put (" + ");
    Float_IO.Put (Item.IP, 5, 5, 0);
    Text_IO.Put ("j");
  end Print;

  procedure Print_All is
```

```
  begin
    Text_IO.Put ("N1   = "); Print (N1); Text_IO.New_Line;
    Text_IO.Put ("  N2 = "); Print (N2); Text_IO.New_Line;
    Text_IO.Put ("  N3 = "); Print (N3); Text_IO.New_Line;
  end Print_All;

begin -- Main

  Print_All;
  N1 := (2.2, 4.4);
  N2 := (1.0, 12.2);
  N3 := N1 + N2;
  Text_IO.Put_Line ("Sum:");
  Print_All;
  N3 := N1 - N2;
  Text_IO.Put_Line ("Difference:");
  Print_All;
  N3 := N1 * N2;
  Text_IO.Put_Line ("Product:");
  Print_All;

end Main;
```

§

Output

```
µN1   =     0.00000 +     0.00000j
  N2 =     0.00000 +     0.00000j
  N3 =     0.00000 +     0.00000j
Sum:
N1   =     2.20000 +     4.40000j
  N2 =     1.00000 +    12.20000j
  N3 =     3.20000 +    16.60000j
Difference:
N1   =     2.20000 +     4.40000j
  N2 =     1.00000 +    12.20000j
  N3 =     1.20000 +    -7.80000j
Product:
N1   =     2.20000 +     4.40000j
  N2 =     1.00000 +    12.20000j
  N3 =     3.84000 +    31.24000j
```

§

Solution 4.2:

Code

```
µ-- Problem 4.2
-- by Rick Conn
package Complex is

  type NUMBER is private;

  function Set (RP, IP : in FLOAT) return NUMBER;
  function "+" (Left, Right : in NUMBER) return NUMBER;
  function "-" (Left, Right : in NUMBER) return NUMBER;
  function "*" (Left, Right : in NUMBER) return NUMBER;
  procedure Print (Item : in NUMBER);

private

  type NUMBER is record
    RP : FLOAT := 0.0;  -- initialize to 0,0
    IP : FLOAT := 0.0;
  end record;

end Complex;

with Text_IO;
package body Complex is

  package Float_IO is new Text_IO.Float_IO (FLOAT);

  function Set (RP, IP : in FLOAT) return NUMBER is
  begin
    return (RP, IP);
  end Set;

  function "+" (Left, Right : in NUMBER) return NUMBER is
  begin
    return (Left.RP + Right.RP, Left.IP + Right.IP);
  end "+";

  function "-" (Left, Right : in NUMBER) return NUMBER is
  begin
    return (Left.RP - Right.RP, Left.IP - Right.IP);
  end "-";

  function "*" (Left, Right : in NUMBER) return NUMBER is
```

```
  begin
    return (Left.RP * Left.RP - Right.RP * Right.RP,
            Left.RP * Right.IP + Left.IP * Right.RP);
  end "*";

  procedure Print (Item : in NUMBER) is
  begin
    Float_IO.Put (Item.RP, 5, 5, 0);
    Text_IO.Put (" + ");
    Float_IO.Put (Item.IP, 5, 5, 0);
    Text_IO.Put ("j");
  end Print;

end Complex;

with Text_IO;
with Complex;
use Complex;  -- so infix operators may be used
procedure Main is

  N1, N2, N3 : Complex.NUMBER;

  procedure Print_All is
  begin
    Text_IO.Put ("N1   = "); Complex.Print (N1); Text_IO.New_Line;
    Text_IO.Put ("  N2 = "); Complex.Print (N2); Text_IO.New_Line;
    Text_IO.Put ("  N3 = "); Complex.Print (N3); Text_IO.New_Line;
  end Print_All;

begin -- Main

  Print_All;
  N1 := Set (2.2, 4.4);
  N2 := Set (1.0, 12.2);
  N3 := N1 + N2;
  Text_IO.Put_Line ("Sum:");
  Print_All;
  N3 := N1 - N2;
  Text_IO.Put_Line ("Difference:");
  Print_All;
  N3 := N1 * N2;
  Text_IO.Put_Line ("Product:");
  Print_All;

end Main;
```

```
§
```

Output

```
μN1  =    0.00000 +    0.00000j
  N2 =    0.00000 +    0.00000j
  N3 =    0.00000 +    0.00000j
Sum:
N1   =    2.20000 +    4.40000j
  N2 =    1.00000 +   12.20000j
  N3 =    3.20000 +   16.60000j
Difference:
N1   =    2.20000 +    4.40000j
  N2 =    1.00000 +   12.20000j
  N3 =    1.20000 +   -7.80000j
Product:
N1   =    2.20000 +    4.40000j
  N2 =    1.00000 +   12.20000j
  N3 =    3.84000 +   31.24000j

§
```

Solution 4.3:

Code

```
µ-- Problem 4.3
-- by Rick Conn
generic
  type ELEMENT is private;
  type INDEX is (<>);
  type VECTOR is array (INDEX range <>) of ELEMENT;
  with function "<" (Left, Right : in ELEMENT) return BOOLEAN;
procedure Bubble_Sort (Item : in out VECTOR);

procedure Bubble_Sort (Item : in out VECTOR) is
  Temp : ELEMENT;
begin
  for I in Item'RANGE loop
    for J in INDEX'SUCC(I) .. Item'LAST loop
      if Item(J) < Item(I) then
        Temp    := Item(I);
        Item(I) := Item(J);
        Item(J) := Temp;
      end if;
    end loop;
  end loop;
end Bubble_Sort;

with Text_IO;
with Bubble_Sort;
procedure Main is

  type REC is record
    N1 : INTEGER;
    N2 : INTEGER;
  end record;

  type COLOR is (RED, GREEN, BLUE, YELLOW, VIOLET, GRAY, WHITE);

  type RECARRAY is array (COLOR range <>) of REC;
  type INTARRAY is array (NATURAL range <>) of INTEGER;
  type FLTARRAY is array (NATURAL range <>) of FLOAT;

  R1 : RECARRAY(RED .. YELLOW) := ((5, 2), (2, 2), (10, 10), (3, 3));
  I1 : INTARRAY(5..9) := (5, 2, 6, 1, 3);
  F1 : FLTARRAY(11..12) := (2.2, 1.1);
```

```
function "<" (Left, Right : in REC) return BOOLEAN is
begin
  return Left.N1 + Left.N2 < Right.N1 + Right.N2;
end "<";

procedure Int_Sort is new Bubble_Sort
  (ELEMENT => INTEGER,
   INDEX   => NATURAL,
   VECTOR  => INTARRAY,
   "<"     => Standard."<");

procedure Float_Sort is new Bubble_Sort
  (ELEMENT => FLOAT,
   INDEX   => NATURAL,
   VECTOR  => FLTARRAY,
   "<"     => Standard."<");

procedure Rec_Sort is new Bubble_Sort
  (ELEMENT => REC,
   INDEX   => COLOR,
   VECTOR  => RECARRAY,
   "<"     => Main."<");

package Int_IO is new Text_IO.Integer_IO (INTEGER);
package Flt_IO is new Text_IO.Float_IO (FLOAT);

procedure Print_All is

  procedure Rec_Print (R : in REC) is
  begin
    Text_IO.Put("  ("); Int_IO.Put (R.N1, 3); Text_IO.Put(", ");
    Int_IO.Put (R.N2, 3); Text_IO.Put (')');
  end Rec_Print;

begin
  Text_IO.Put ("Records: ");
  for I in R1'RANGE loop
    Rec_Print (R1(I));
  end loop;
  Text_IO.New_Line;
  Text_IO.Put ("Integers: ");
  for I in I1'RANGE loop
    Int_IO.Put (I1(I), 3);
  end loop;
  Text_IO.New_Line;
```

```
   Text_IO.Put ("Floats: ");
   for I in F1'RANGE loop
     Flt_IO.Put (F1(I), 3, 1, 0);
   end loop;
   Text_IO.New_Line;
  end Print_All;

begin -- Main

  Print_All;
  Rec_Sort (R1);
  Int_Sort (I1);
  Float_Sort (F1);
  Print_All;

end Main;
```

§

Output

```
µRecords:  ( 5,  2) ( 2,  2) ( 10,  10) ( 3,  3)
Integers:  5  2  6  1  3
Floats:  2.2  1.1
Records:  ( 2,  2) ( 3,  3) ( 5,  2) ( 10,  10)
Integers:  1  2  3  5  6
Floats:  1.1  2.2
```

§

Solution 4.4:

Code

```
µ-- Problem 4.4
-- by Rick Conn
with Text_IO;
procedure Main is

  task type Display is
    entry Start (Task_Name : in STRING;
                 Delay_Amt : in DURATION);
  end Display;

  task body Display is
    T_Name       : STRING(1..80);
    TNLast       : NATURAL;
    Delay_Amount : DURATION;
  begin
    accept Start (Task_Name : in STRING;
                  Delay_Amt : in DURATION) do
      T_Name (1..Task_Name'LENGTH) := Task_Name;
      TNLast                       := Task_Name'LENGTH;
      Delay_Amount                 := Delay_Amt;
      Text_IO.Put_Line ("Task " & T_Name(1..TNLast) & " starting");
    end Start;
    delay Delay_Amount;
    Text_IO.Put_Line ("Task " & T_Name(1..TNLast) & " ending");
  end Display;

begin -- Main

  declare
    Tasks : array (1..5) of Display;
  begin
    for I in Tasks'RANGE loop
      Tasks(I).Start
         ("Task" & INTEGER'IMAGE(I), DURATION(Tasks'LAST+1 - I));
    end loop;
  end;

end Main;
```

§

Output

```
µTask Task 1 starting
Task Task 2 starting
Task Task 3 starting
Task Task 4 starting
Task Task 5 starting
Task Task 5 ending
Task Task 4 ending
Task Task 3 ending
Task Task 2 ending
Task Task 1 ending

§
```

Solution 4.5:

Code

```
µ-- Problem 4.5
-- by Rick Conn
with Text_IO;
procedure Main is

  DIVIDE_BY_ZERO : exception;

  type INTARRAY is array (INTEGER range <>) of INTEGER;

  I1 : constant INTARRAY := (2, 3, 4, 5, 6, 7);
  J1 : constant INTARRAY := (2, 0, 1, 0);
  K  : INTEGER;

  package Int_IO is new Text_IO.Integer_IO (INTEGER);

  function "/" (Left, Right : in INTEGER) return INTEGER is
  begin
    if Right = 0 then
      raise DIVIDE_BY_ZERO;
    end if;
    return Standard."/" (Left, Right);
  end "/";

  procedure Print_ALL (I1_Index, J1_Index, K : in INTEGER) is
  begin
    Int_IO.Put (I1(I1_Index), 2);
    Text_IO.Put ("/");
    Int_IO.Put (J1(J1_Index), 2);
    Text_IO.Put (" = ");
    Int_IO.Put (K, 2);
    Text_IO.New_Line;
  end Print_All;

begin -- main

  for J in J1'RANGE loop
    for I in I1'RANGE loop
      begin
        K := I1(I) / J1(J);
        Print_All (I, J, K);
      exception
        when DIVIDE_BY_ZERO =>
```

```
        Text_IO.Put ("Division by Zero: ");
        Int_IO.Put (I1(I), 2);
        Text_IO.Put (" / ");
        Int_IO.Put (J1(J), 2);
        Text_IO.New_Line;
     end;
   end loop;
 end loop;

end Main;
```

§

Output

```
μ 2/ 2 =  1
 3/ 2 =  1
 4/ 2 =  2
 5/ 2 =  2
 6/ 2 =  3
 7/ 2 =  3
Division by Zero:  2 /  0
Division by Zero:  3 /  0
Division by Zero:  4 /  0
Division by Zero:  5 /  0
Division by Zero:  6 /  0
Division by Zero:  7 /  0
 2/ 1 =  2
 3/ 1 =  3
 4/ 1 =  4
 5/ 1 =  5
 6/ 1 =  6
 7/ 1 =  7
Division by Zero:  2 /  0
Division by Zero:  3 /  0
Division by Zero:  4 /  0
Division by Zero:  5 /  0
Division by Zero:  6 /  0
Division by Zero:  7 /  0
```

§