ADA95 Tutorial
Part 1

produit à partir du matériel disponible à l'URL http://www.infres.enst.fr/~pautet/Ada95/a95list.htm

ADA 95 TUTORIAL

This tutorial teaches the entire Ada 95 dialect of the Ada language. It is composed of 33 chapters which should be studied in order since topics are introduced in a logical order and build upon topics introduced in previous chapters. It is to the students benefit to download the source code for the example programs, then compile and execute each program as it is studied. The diligent student will modify the example program in some way, then recompile and execute it to see if he understands the material studied for that program. This will provide the student with valuable experience using his compiler.

The recommended method of study is to print the text for one or two chapters, download the example programs, and study the material by loading the example programs in the compiler's editor for viewing. Following successful completion of each chapter, additional chapters can be downloaded as progress is made.

Note that completion of the first part of this tutorial will give the student the ability to write very significant programs in Ada, but completion of the second part will give the student the ability to use all of the capabilities of Ada.

Version 2.5 - February 1, 1998

The original for this page is located at http://www.swcp.com/~dodrill/a95doc/a95list.htm and is the only fully authorized site for distribution of this tutorial. Many persons have downloaded one or more of our tutorials for redistribution without our consent, and occasionally do not include all of the components needed for the complete package. You can be assured that the tutorial will be complete and up to date only at the home site, since we have no control over the actions of other web site operators.

This tutorial is distributed as shareware which means that you do not have to pay to use it. However, the author spent a good deal of time and financial resources to develop this tutorial and requests that you share in the financial burden in a very small way, but only if you felt the tutorial was valuable to you as an aid in learning to program in Ada. If you wish to remit a small payment to the author, full instructions for doing so will be given by clicking the link below. I hope you find programming in Ada to be rewarding and profitable. I personally think Ada is the best language to use for a large project with more than a single programmer because of the careful interface checking done by the compiler.

How to Remit Payment For this Tutorial!

Part 1 - Beginning Ada 95 Tutorial

Ada Tutorial - Chapter 1

GETTING STARTED

WHAT IS ADA?

Ada is a relatively new programming language developed by the United States Department of Defense in an attempt to solve the software muddle as it existed in the mid 1970's. It was felt that the 2000 or so programming languages in use at that time could be replaced in large part by one well planned language for use in embedded Real-Time systems. Following a major effort on the part of the DOD, which is well documented in many other places, Ada was developed as a solution to the software problem.

WHAT IS ADA 95?

Ada 95 is an ISO update to the Ada programming language to incorporate the latest knowledge of software development into the language. The 1983 version of the language has been called Ada for years. Since the newer version is meant to eventually supersede, and displace the original version, the newer version is also referred to as Ada. It is therefore up to the reader to determine which version of Ada the author is discussing by the context or by a direct statement of intent. To make it as simple as possible, this tutorial will use the terms Ada and Ada 95 interchangeably when discussing the newer version of the language, and all references to the original language will be made with Ada 83. Any place where they are being compared, the text will use the full name, Ada 95, to alleviate any possible confusion.

Ada is a very well planned and precisely defined language that can be used throughout a wide area of software applications. The language has existed long enough that a reasonable number of capable compilers exist for use on mainframe computers, as well as minicomputers, and even microcomputers. An Ada compiler has a big job to do which you will see as we progress through our study of the language. It is therefore not a trivial effort to bring a validated Ada compiler to market. In spite of this, at least three companies have developed fully validated Ada compilers that run under MS-DOS and/or Windows on a PC. Although some of these will run on a minimal PC, a relatively powerful PC is recommended for use with any Ada compiler due to the time required for compilation.

The Ada programming language was designed in such a way that many of the trivial errors, which we humans are very capable of generating, are detected and reported at compile time rather than after execution of the program is begun. It is at this point that errors are most easily repaired since a good compiler can give the programmer a very good hint at just what the error is.

This chapter will give you some definitions so we can begin discussing the use of Ada in chapter 2. The definitions will be very broad in nature because they are used in many places in an Ada program, but they are extremely important.

WHAT IS AN IDENTIFIER?

An identifier is a name we use to refer to any object in Ada and it must be formed by following some fairly rigid rules. We will list the rules for forming a valid identifier, then make up a few for illustrative purposes.

1. An identifier must start with a letter of the alphabet.
2. Following the initial letter, the identifier can be made up of as many letters, numbers, and underlines as desired provided that the underlines occur only singly, and an underline is not the last character.
3. Case of letters is not significant.
4. There is no limit to the length of an identifier but each identifier must fit on one line of text and the writer of the compiler may impose a line length limit. The minimum line length must be at least 200 characters.

5. No blanks or special characters can be used as part of an identifier.

With these rules in mind, lets make up a few good identifiers and a few invalid identifiers.

```
Ada            -- A perfectly valid identifier
ADA            -- The same one, case doesn't matter
Ada_Compiler   -- A very descriptive identifier
The_Year_1776  -- Another descriptive identifier
a1b2c3d4e5f6   -- Very nondescript, but valid

12_times_each  -- Can't start with a number
This__is__neat -- Multiple underlines illegal
This is neat   -- blanks illegal
Ada_"tutorial" -- special characters illegal
```

By this time you should get the idea of what a valid Ada identifier is. It may seem like a lot of effort to define just what an identifier is, but you will be very busy naming everything you use in Ada, so you must know how to name things before you can do anything meaningful with the language.

IDENTIFIER SELECTION

In addition to an identifier being correct, it should also be usable and meaningful. As an example, consider the following list of valid identifiers and see which convey to you some idea of what they refer to.

```
Time_Of_Day
Final_Score
Get_the_Present_Temperature
X12
Ztx
t
```

Ada was designed to be written once and read many times. This is truly what happens with any non-trivial program designed and developed by a group of persons. As such, little attention is paid to the fact that it may be a bit tedious to key in long identifiers when the program is being written. The extra effort pays off when it is read repeatedly, since it is so easy to follow the logic of the program. The first three identifiers above are preferred because of the information they convey to the reader, and the last three are to be considered of little value in defining the program logic. Of course, if you were using a mathematical relationship that used the variable named "t" in its calculations, that particular name for a variable might be a good choice.

WHAT ARE RESERVED WORDS?

Ada 95 uses 69 identifiers which are called reserved words. Note that Ada 83, by contrast, only had 63 reserved words. Reserved words are reserved for specific uses within an Ada program and cannot be used for any other purpose. As you study the language, you will see very clearly how to use each of the reserved words and why these particular words were chosen. Since Ada is a large language containing many options and cross checks, writing an Ada compiler is an enormous job, but the use of reserved words simplifies that job. The reserved words also make the final program much easier to read and understand.

Don't worry about the reserved words at this point. It was necessary to mention that they do exist and constitute an additional limitation to the naming of identifiers which we discussed in the previous section. It might be a good idea to spend a few minutes looking through the list in section 2.9 of the Ada 95 Reference Manual (ARM). Note that all reserved words will be listed in **boldface** type when used in the text portion of this tutorial.

CASE CONVENTIONS USED IN THIS TUTORIAL

Ada allows you to use either case for alphabetic characters in an identifier and you can freely mix them up in any way you desire. Good programming practice, however, would lead you to select a convention for where to use upper case and where to use lower case. A good selection of case could be an aid to understanding the program since it would convey some information about what the identifier is.

In order to write the example programs in a standard format, the author did a search of Ada programs to see if a standard exists which would dictate which case should be used for alphabetic characters. The search was conducted by studying the code in the books mentioned in the Introduction to this tutorial and about 12 other books. No conformance to any standard was found, so the following will be adopted for all of the sample programs in this tutorial. Since you are just beginning to study Ada, you may not understand what each of the categories are. After you complete a few of the lessons, you can return here to review the alphabetic case rules listed for the example programs.

reserved words - All reserved words will be written in lower case. This is the only consistency found in the search of the Ada programs.

Variables - All variables will be written with the initial letter of each word capitalized, and all others in lower case.

TYPES - All types will be written in all capital letters.

CONSTANTS - All constants will be written in all capital letters.

ENUM VALUES - All enumerated values will be written in all capital letters.

ATTRIBUTES - All attributes will be written in all capital letters.

Procedure Names - All procedure names will be written with the initial letter of each word capitalized and all others in lower case.

Function Names - Same as procedure names.

Package Names - Same as procedure names.

Library Names - Same as procedure names.

Note that all program identifiers will be listed in **boldface type** when they are referred to in the text portion of this tutorial.

WHAT ABOUT PROGRAMMING STYLE?

Programming style can go a long way to aiding in the understanding of a completed program and much discussion throughout this tutorial will be given to style. You have the freedom to add indentation and blank lines to your program in your own way to improve readability and at the same time make the program look like your own work. In the early lessons, however, it would be to your advantage to follow the style given in the example programs and adopt it as your own. As you gain experience, you will develop your own style of Ada source code formatting.

PRELIMINARY DEFINITIONS

Several topics, which are unique to Ada, are used in many places throughout the language. Since a full definition of these will be impossible until we cover some of the earlier topics, we must delay the full definition until later. On the other hand, the use of them becomes necessary fairly soon, so we will give a brief definition of these now, and a complete definition later. If you don't fully understand these early definitions, don't worry about it, because we will return for a fuller definition later.

Exceptions - When most languages find a fatal runtime error, they simply abort the program. This is unacceptable for a real time language because it must continue running, correcting the error if possible. An exception is an exceptional, or error, condition that arises during execution of the program. An Ada program, if it is properly written, has the ability to define what to do for each of these error conditions, and continue operation.

Renaming - Ada gives you, the programmer, the ability to assign a new name to various entities in a program for your own convenience. Ada permits the renaming of objects, exceptions, task entries, and subprograms. It is simply an alias which can be used to refer to the entity which is renamed.

Overloading - Ada allows you to use the same name for several different items. The system is smart enough to know which entity you are referring to, when you use the overloaded name, by the immediate context of its use. For example, if I say, "Jack used a jack to change the flat tire.", you understand that there are two uses of the word "Jack", and you understand what each means by the way it is used in the statement. Ada can also use the same name to refer to different things, and intelligently know what the various uses mean. We will revisit this topic later in this tutorial.

Ada Tutorial - Chapter 2

PROGRAM STRUCTURE

OUR FIRST ADA PROGRAM

Example program ------> **e_c02_p1.ada**

```
 with Ada.Text_IO     ;         use Ada.Text_IO;procedure UglyForm
is begin Put   ("Good form ")                    ;Put("can aid in ")
;Put  ("understanding a program,");New_Line;Put
("and bad form ");Put   ("can make a program ");Put("unreadable.");
   New_Line;end UglyForm;




-- Result of execution

-- Good form can aid in understanding a program,
-- and bad form can make a program unreadable.
```

The best way to begin our study of Ada is to look at a real Ada program, so by whatever means you have at your disposal, display the Ada program named e_c02_p1.ada on your monitor.

You are looking at the simplest Ada program that it is possible to write. Even though it is the simplest, it is a complete executable program that you will be able to compile and execute after we discuss a few of the entities that appear in it.

The word **procedure** in line 1 is the first of the reserved words we will look at. Until we reach a much later point in this tutorial, we will simply say that a procedure is a program. In this case, therefore, the program is defined by the reserved word **procedure** followed by the program name and another reserved word **is**. Following the reserved word **is**, we have the actual program extending to line 5. The reserved word **is** therefore, is saying that the program (or procedure) which is named **Trivial** is defined as everything that follows.

The program name, in this case **Trivial**, must be selected by following all of rules given in chapter 1 for naming an identifier. In addition to those rules, it cannot be one of the 69 reserved words.

WHERE IS THE ACTUAL PROGRAM?

There are two portions of any Ada program, a declarative part, which is contained between the reserved words **is** and **begin**, and the executable part which is contained between the reserved words **begin** and **end**. In this particular case, there is nothing in the declaration part, and the executable part consists of line 4. (We will return to line 4 shortly.) Following the reserved word **end** is a repeat of the program name and a semicolon. Repeating the program name is optional but, as a matter of style, you should get into the habit of including it at the end of every program. The semicolon is required to end the program.

The actual program, the executable part, is line 4 since that is the only statement between the **begin** and **end** reserved words. In this case we wanted to study the simplest Ada program possible so we wanted the program to do nothing. We explicitly tell the Ada compiler to do nothing which is the definition of our fifth reserved word **null** in line 4. Ada demands that you explicitly tell it that you really mean to do nothing rather than simply leaving the executable part of the program blank which would be acceptable in most other languages.

WHAT IS A STATEMENT TERMINATOR?

Line 4 ends with a semicolon which is a statement terminator in Ada. It tells the compiler that this particular statement is complete at this point. Later in this chapter you will see why the statement terminator is required. The semicolon at the end of the procedure is also a statement terminator since a procedure, and hence the entire program, is defined as a complete statement in Ada.

Lines 10 and 12 are Ada comments and will be ignored by the compiler. A complete definition of Ada comments will be given at the end of this chapter. All example programs in this tutorial will give you the results of execution at the end of the program as illustrated here.

It should be clear that a complete Ada program uses at least the four reserved words to define the beginning and end of each of the fields and they must come in the given order. Of course many other things can be included in the declarative part and the executable part which we will see during the remainder of this tutorial.

The program outline can be given as follows;

```
procedure <program name> is
   <declarative part>
begin
   <executable part>
end <optional repeat of program name>;
```

At this point it would be wise for you to compile and run this program to see that it truly does obey all the rules of the Ada compiler. Unfortunately, it doesn't do anything, so running it will give you no results. Even though this program does nothing, any good Ada compiler will allow you to compile, link, load, execute, and properly terminate execution of this program.

NOW FOR A PROGRAM THAT DOES SOMETHING

Example program ------> **e_c02_p1.ada**

Observe the program named e_c02_p1.ada for an example program that really does something. Ignore the first two lines for the time being, they are needed to tell the system how to output data to the monitor. We will study them in a later lesson. Considering only lines 4 through 10, you will see exactly the same structure used in the last program with a different program name and something new in line 8.

Line 8 is a call to a procedure named **Put** which is supplied with your Ada compiler as a convenience for you. It is very precisely defined so that you can use it to display text on your monitor. The string of characters contained within the parentheses and quotation marks will be displayed on your monitor when you compile and execute this program. The procedure named **Put** is actually a part of a library named **Ada.Text_IO** which is why the first two lines are included in this program. They tell the system where to find the procedure named **Put**, and how to use it. (We will discuss these two lines in great detail later in this tutorial.)

Once again, the executable statement in line 8 has a semicolon at the end as the statement terminator in the same manner that the reserved word **null** was followed by a semicolon in the last program.

Compile and execute this program and observe that the phrase in line 8 is displayed on the monitor each time you execute the program.

A LITTLE MORE OUTPUT

Example program ------> **e_c02_p1.ada**

Examine the program named e_c02_p1.ada, and you will see a few differences from the last two example programs. Observe that the program name is not repeated in the last line of the program. This is optional as we stated earlier, but it is a good practice to include the name.

The second, and most obvious difference, is the fact that there are several executable statements in this program. The executable statements, as with most other procedural programming languages, are executed in sequential order from top to bottom. The lines with calls to the procedure **Put** are executed just like the last program except a new operation becomes apparent here because we have multiple **Put** calls. The **Put** call does not cause the cursor to return to the beginning of a line following output of the line of text. The cursor simply stays where it ends up at, resulting in lines 8 and 9 being output on the same line of the monitor. Another new procedure, at least new to us, is named **New_Line** and this procedure causes the cursor to be returned to the beginning of the next line of the monitor. In fact, when using the **New_Line** procedure, you can even include an optional number within parentheses following the procedure name, and the cursor will be moved down that number of lines. This is illustrated in lines 12 and 14, and if the optional number is omitted, a value of 1 is assumed.

Note carefully that the procedure names used here, **Put** and **New_Line**, meet all of the requirements for naming an identifier which we studied in chapter 1. These names were selected by the Ada language definition committee.

Lines 16 and 17 introduce another useful procedure, named **Put_Line**, which causes an automatic "carriage return" to be output after the string within the parentheses is output. You will find this to be very useful, and should begin using it immediately. The blank line in line 15 is ignored by the Ada compiler. More will be said about the use of white space in the next two example programs.

HOW DID WE NAME THE IDENTIFIERS?

We have the option of naming our program any name we wish as long as we obey all of the rules of naming an identifier listed in Chapter 1 of this tutorial. We have a restriction on the program name because of the way Ada compiles and links a program. In order to meet all of the requirements as specified in the Ada 95 Reference Manual (ARM), a compiler must generate some form of intermediate files containing object and type information. Any particular Ada compiler may use a naming convention for the intermediate files based on the program name we supply, or the file name we supply, so in order to avoid confusion, the same name will be used in both places throughout most of this tutorial. Therefore the name of the procedure, **MoreOut** in this case, will be the same as the name of the file, e_c02_p1.ada in this case. Note that your particular Ada compiler may not have this limitation.

With the above information, you should be able to figure out what this program will do. Compile and run it to see if you are correct in your analysis. This is a great place for you to begin programming by using these three subprograms to output some text in a neat format.

CONSIDER THE STYLE OF ADA PROGRAMMING

Example program ------> **e_c02_p1.ada**

Observe the program named e_c02_p1.ada and you will see a familiar form, and a very clear and easy to understand Ada program. You should have no problem at all understanding what this program does. You should observe that Ada programming is free form, allowing you to add spaces and blank lines anywhere you wish to make the program clear and easy to understand, provided of course that you do not split up an identifier.

Once again, don't worry about the first two lines, we will discuss them later.

Example program ------> **e_c02_p1.ada**

After compiling and executing e_c02_p1.ada, observe the next example program named e_c02_p1.ada for an example of terrible formatting style. See if you can figure out what this program does, then decide if you would like to debug it if a problem should surface. Finally, compile and execute this program and you will find that it actually does compile and execute, doing exactly what the last program did.

These two programs were intended to give you an idea of the amount of freedom you have in formatting style when writing an Ada program and the amount of information the style can add to a program.

COMMENTS IN AN ADA PROGRAM

Example program ------> **e_c02_p1.ada**

Examine the program named e_c02_p1.ada for an example of how comments are added to an Ada program. Comments convey no information to the computer, they only aid the writer and reader of the program to understand what the original writer was trying to do within the program. All comments in Ada begin with a double minus sign, or double dash if you prefer, and continue to the end of that line. No spaces are allowed between the dashes, they must be adjacent. There is no provision for middle-of-line comments in the Ada language, only end-of-line, although they can be an entire line as illustrated in the first two lines of the example program.

Comments can be placed nearly anywhere in an Ada program, including prior to the program, and following the end of it. The example program gives many illustrations of where comments can go and it will be left to your study. Note that line 16 is not an executable statement since it is commented out. As with all programs in this tutorial, this one is executable, so you should compile and execute it at this time.

One other point must be made. This example program is not meant to be an illustration of good commenting style, only an indication of where comments can go. It is actually a very choppy looking program, and is not at all clear.

PROGRAMMING EXERCISES

1. Write a program to display your name on the monitor.(Solution)

```
                           -- Chapter 2 - Programming Exercise 1
with Ada.Text_IO;
use Ada.Text_IO;

procedure Ch02_1 is

begin
   Put("John Q. Doe");
end Ch02_1;




-- Result of execution

-- John Q. Doe
```

2. Write a program to display your name, address, and phone number on different lines of the monitor.(Solution)

```
                           -- Chapter 2 - Programming Exercise 2
with Ada.Text_IO;
use Ada.Text_IO;

procedure Ch02_2 is

begin
   Put("John Q. Doe");
   New_Line;
   Put("Anywhere, Anystate, USA, 12345");
   New_Line;
   Put("(123) 456-7890");
   New_Line;
end Ch02_2;
```

```
-- Result of execution

-- John Q. Doe
-- Anywhere, Anystate, USA, 12345
-- (123) 456-7890
```

Ada Tutorial - Chapter 3

THE INTEGER TYPE VARIABLE

OUR FIRST INTEGER VARIABLE

Example program ------> **e_c03_p1.ada**

```
                                       -- Chapter 3 - Program 1
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure OneInt is

   Index : INTEGER;           -- A simple Integer type

begin

   Index := 23;
   Put("The value of Index is");
   Put(Index);                -- The default field width is 11 columns
   New_Line;
   Index := Index + 12;
   Put("The value of Index is");
   Put(Index, 8);
   New_Line;

end OneInt;




-- Result of execution

-- The value of Index is         23
-- The value of Index is      35
```

Examine the program named e_c03_p1.ada for our first example program with a variable. Some programming languages do not require you to predefine a variable before you use it. Instead, you simply begin using it and the system has some mechanism by which it creates the variable and makes it ready for your use. Ada requires you to specifically define every variable before you use it. You must give the variable a name, which is any valid identifier, and you tell the compiler how you plan to use the variable by assigning a type to the variable.

WHAT IS A TYPE?

The type defines a set of values which the variable can have assigned to it, and it also defines a set of operations that can be performed on the variable. Ada is a strongly typed language since it has very strict rules limiting how a variable can be used. The compiler will not give you a usable program unless you follow all of the rules very carefully. A major part of the study of Ada involves the study of types and how to use typing as a programming aid.

AN UNBROKEN RULE OF ADA, NEVER BROKEN

Ada requires that anything you use must have been previously defined. This includes variables as well as constants, procedures, functions, and all other entities. At least one language, Pascal, makes this claim but breaks it in a few instances. Ada never breaks this rule.

THE with AND use STATEMENTS

The **with** and **use** statements, in lines 2 and 3 of this program, contain **Ada.Integer_Text_IO** in addition to **Ada.Text_IO** and a few comments are in order at this point, event though they will be completely defined later in this tutorial. The term **Ada.Text_IO** refers to an Ada package of the same name that provides us with the ability to output text to the monitor including characters, strings, carriage returns, and various other entities so that we can generate some formatted text output to the monitor. It also gives us the ability to input text from the keyboard, and it provides some file input/output capabilities.

The term **Ada.Integer_Text_IO** refers to another Ada package of that name which gives us the ability to output **INTEGER** type variables to the monitor in a well formatted way since it gives us control over how many columns are used, and what base to use for the numbering system, such as binary, decimal, hexadecimal, and other options. The **with** statement, because it mentions both of these library packages, provides us with the ability to output both text and numeric values to the monitor, because it makes a copy of both of these libraries available for use by our program. The **use** statement, because it mentions both of these packages, makes it very easy to "use" these packages within our program. Without the **use** statement, we would have to qualify every input or output statement, which makes for very ugly code, but which some programers find preferable for various reasons.

We will cover all of this in great detail later in this tutorial, but it would be best for you to simply include the appropriate packages in a **with** statement and a **use** statement for your initial programming efforts and pick up the additional knowledge later. There is only so much you can absorb and understand at one time, and the author of this tutorial desires to simplify your study of Ada as much as possible by deferring some topics until later.

HOW DO WE DECLARE A VARIABLE?

Back to the program named e_c03_p1.ada. To understand the definition of a variable, examine the program at hand and specifically line 7. This declares a variable, which we will call **Index**, to be of type **INTEGER**, therefore defining an allowable range of values which can be assigned to it. Most 16 bit computers allow an **INTEGER** type variable to cover a range of -32,768 to 32,767. The corresponding range for 32 bit computers is from -2,147,483,648 to 2,147,483,647 but the definition of Ada allows for flexibility in both of these ranges. We will see a way to determine exactly what the limits are for your compiler later in this chapter.

The type also defines a number of operations which can be performed on the variable. More will be said of that later. The type **INTEGER** is used to declare a scalar variable, which is a variable that can contain a single value.

The word **INTEGER** is not a reserved word but a predefined word which we can redefine if we so choose. We will not be doing that for a long time since it could cause untold problems in a program. You should know that it is possible for you to redefine this word, and many other similarly predefined words, to mean something entirely different from their predefined meanings.

In the last chapter, we said that the declarative part of the program goes between the reserved words **is** and **begin**, and you will notice that we did indeed declare the variable named **Index** in that area of the program. The end result of line 7 is that we have a variable named **Index** which is of type **INTEGER**. As yet however, it does not contain a useful value.

HOW DO WE USE THE VARIABLE IN A PROGRAM?

In the last chapter we also said that the executable statements went between the **begin** and the **end** reserved words and you will see that we have some executable statements within that range, in lines 11 through 18 of this program. In line 11 we assign the value of 23 to the variable **Index** which is valid to do because 23 is within the range of allowable values that can be assigned to an **INTEGER** type variable.

THE ASSIGNMENT OPERATOR

The combination := can be read as "gets the value of". Line 11 can then be read as, "**Index** gets the value of 23." The equal sign alone is reserved for another use. Actually, if we say that **Index** = 23, it is only mathematically correct if **Index** is never changed, but since it is a variable and will be changed, the equality is not true.

We have succeeded in assigning a value of 23 to the variable named **Index** and can use it in many different ways in an Ada program, but we will illustrate only very simple uses at this point.

Line 12 instructs the system to output a line of text to the monitor and leave the cursor at the end of the line. In line 13, we use the predefined procedure named **Put** to tell the system to display the value of **Index**, which has the value of 23, on the monitor. The system will right justify the output in a field width that depends on the size of an **INTEGER** on your system, because of its definition. This **Put** is part of the **Ada.Integer_Text_IO** package we declared earlier. The **Put** in line 12 is from **Ada.Text_IO**. Later in this tutorial you will understand which **Put** is from which package and why. Line 14 returns the cursor to the beginning of the next line. The **New_Line** procedure is from the **Ada.Text_IO** package.

OUR FIRST ARITHMETIC

Line 15 contains our first arithmetic statement and it will do exactly what it appears to do, "**Index** gets the value of **Index** with 12 added to it." The variable **Index** should now have a stored value of 23 + 12, or 35, which we verify by telling the system to display the new value of **Index**. The numeral 8 in line 17 tells the system to display the value right justified in a field 8 columns wide. We will discuss the **Put** procedure in detail later in this tutorial. If you remember that the statements are executed sequentially, you should have no difficulty following this program.

Compile and execute this program being careful to observe that the two values of **Index** are displayed in fields of different widths.

LET'S USE LOTS OF INTEGERS NOW

Example program ------> **e_c03_p2.ada**

```
                                       -- Chapter 3 - Program 2
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure MoreInts is

   Index_1                    : INTEGER;
   Index_2, Index_3, Index_4  : INTEGER;
   Cat                        : INTEGER := 12;
   Dog                        : INTEGER := -5;
   Pig, Hog, Sow              : INTEGER := 1000;

begin

   Index_1 := Cat + 4;                 -- Index_1 is 16
   Index_2 := Dog - 3;                 -- Index_2 is -8
   Index_3 := Pig * 7;                 -- Index_3 is 7000
   Index_4 := Pig / 300;               -- Index_4 is 3
   Put("Index_1 = "); Put(Index_1); New_Line;
   Put("Index_2 = "); Put(Index_2); New_Line;
   Put("Index_3 = "); Put(Index_3); New_Line;
   Put("Index_4 = "); Put(Index_4); New_Line(2);

   Index_1 := 5 * Cat - Pig / 4 ;      -- Index_1 is -190
   Index_2 := (5 * Cat) - (Pig / 4);   -- Index_2 is -190
   Index_3 := Pig mod 3;               -- Index_3 is 1
   Index_4 := Pig rem 3;               -- Index_4 is 1
   Put("Index_1 = "); Put(Index_1); New_Line;
   Put("Index_2 = "); Put(Index_2); New_Line;
   Put("Index_3 = "); Put(Index_3); New_Line;
   Put("Index_4 = "); Put(Index_4); New_Line(2);

   Index_1 := abs(Dog);               -- Index_1 is 5
   Index_2 := Cat**3;                 -- Index_2 is 1728
   Index_3 := (Cat-5)**(abs(Dog)-2);  -- Index_3 is 343
   Index_4 := -Index_3;               -- Index_4 is -343
   Put("Index_1 = "); Put(Index_1); New_Line;
   Put("Index_2 = "); Put(Index_2); New_Line;
   Put("Index_3 = "); Put(Index_3); New_Line;
   Put("Index_4 = "); Put(Index_4); New_Line(2);

end MoreInts;




-- Result of execution

-- Index_1 =          16
-- Index_2 =          -8
-- Index_3 =        7000
-- Index_4 =           3
--
-- Index_1 =        -190
-- Index_2 =        -190
-- Index_3 =           1
-- Index_4 =           1
--
-- Index_1 =           5
-- Index_2 =        1728
-- Index_3 =         343
-- Index_4 =        -343
```

Examine the program named e_c03_p2.ada and you will see an example of using many variables of type **INTEGER** in a program. Lines 7 and 8 illustrate that you can define one or more variables on a single line. All four variables are of type **INTEGER** and can be assigned values within the range of integer variables as defined for your particular compiler. Each of these variables have no value associated with them since they were created without an initial value. The executable part of the program can assign a value to each of them. The variable named **Cat** is also an **INTEGER** type variable but after being created, it is assigned an initial value of 12. Likewise **Dog** is created and initialized to a value of -5. It should not come as a surprise to you that the three variables in line 11 are created, and each is assigned an initial value of 1000.

According to the Ada definition, line 11 is merely a shorthand for three different lines with a single variable declaration on each line as far as the Ada compiler is concerned. The same is true of line 8. This is a very subtle point, and has no consequence on the program at hand, but will make a difference later when we commence the study of arrays.

NOW TO EXERCISE SOME OF THOSE VARIABLES

Examining the executable part of the program we find the four arithmetic operations in lines 15 through 18 which should be self explanatory. Note that integer division in line 18 results in truncation, not rounding. The four values are displayed on the monitor in lines 19 through 22 in a format utilizing several statements per line which is simply a matter of style.

Continuing with lines 24 and 25, we have examples of more complex mathematical calculations which should be clear to you. The order of precedence of mathematical operators is given in detail in section 4.5 of the Ada 95 Reference Manual (ARM). The order of precedence is similar to other languages and follows common sense. A discussion of the order of precedence will be given at the end of the next chapter of this tutorial.

Lines 26 and 27 illustrate use of the **mod** and **rem** operators, each of which return the remainder which would be obtained following an integer divide operation. They differ only in the sign when negative numbers are involved and since negative numbers are rare when using these operators, little will be said except to give a brief statement of the differences.

```
    mod - gets the sign of the second operator.
    rem - gets the sign of the first operator.
```

TWO MORE OPERATIONS

After displaying the results, four more values are calculated, the first being the absolute value of the variable **Dog** in line 33. This is not a function, it is an operation defined with the reserved word **abs**. The fact that it is an operator will be very significant when we come to the portion of this tutorial that deals with overloading operators. The **abs** operator returns the absolute value of the variable given to it as a parameter.

The operation in line 34 is an illustration of exponentiation of integer numbers. Since **Cat** has the value of 12, this line says that **Index_2** gets the value of 12 raised to the 3rd power. The only rules for using exponentiation with integer values are, the exponent must be an integer type value, and it cannot be negative. Note that a zero value for an exponent is legal. Line 35 is a combination of several of the previous operations, and line 36 is an illustration of unary negation.

Be sure to compile and execute this program and study the results.

HOW DO WE DECLARE A CONSTANT?

Example program ------> **e_c03_p3.ada**

```
                                       -- Chapter 3 - Program 3
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure InitEx is

   Index_1, Index_2, Index_3    : INTEGER;
   This_Is_A_Long_Variable_Name : INTEGER;

   DOZEN  : constant INTEGER := 12;
   GROSS  : constant INTEGER := 12 * DOZEN;
   BIG_NO : constant         := 32_24_7;          -- This is 32247
   TWO    : constant         := BIG_NO - 3_22_45;  -- This is 2

begin

   Index_1 := GROSS;                    -- Index_1 is 144
   Index_2 := BIG_NO - TWO;             -- Index_2 is 32245
   Index_3 := TWO * GROSS;              -- Index_3 is 288
   This_Is_A_Long_Variable_Name := DOZEN * DOZEN - GROSS;
   Put("Index_1 = "); Put(Index_1); New_Line;
   Put("Index_2 = "); Put(Index_2); New_Line;
   Put("Index_3 = "); Put(Index_3); New_Line;
   Put("This_Is_A_Long_Variable_Name =");
   Put(This_Is_A_Long_Variable_Name); New_Line;

   Index_1 := 123E2;                    -- 12300
   Index_2 := 1_23e2;                   -- 12300
   Index_3 := 12_3e+2;                  -- 12300
   Index_1 := 2#10111#;                 -- Binary number
   Index_2 := 8#377#;                   -- Octal number
   Index_3 := 16#1FF#e1;                -- Hexadecimal number
   Index_1 := 12#A4#;                   -- Base 12 number

end InitEx;




-- Result of execution

-- Index_1 =         144
-- Index_2 =       32245
-- Index_3 =         288
-- This_Is_A_Long_Variable_Name =          0
```

Examine e_c03_p3.ada for an example of a program with some **INTEGER** type variables and some **INTEGER** type constants declared and used in it. Lines 7 and 8 should be familiar to you now, but when we get to lines 10 through 13 we have a few new things to observe. **DOZEN** and **GROSS** are **INTEGER** type constants because of the reserved word **constant** in their declaration. The only difference between a constant and a variable is that a constant cannot be changed during execution of the program and in the present example, it would probably be silly to redefine how many elements are in a dozen. This is one of the things Ada can do to help you if you analyze your program right, because if you ever tried to change the value of **DOZEN** during program execution, Ada would give you an error message and you would eliminate one bug immediately.

Notice that **GROSS** is defined in terms of **DOZEN** since the constant **DOZEN** is available when **GROSS** is initialized. Likewise, the constant **TWO** is defined in terms of **BIG_NO** in the next two lines. It should be obvious that a constant must have an initialization value assigned to it at the point of declaration.

TWO MORE DEFINITIONS

Lines 12 and 13 contain underlines in the numeric values that the Ada compiler will simply ignore. You can put them in wherever you please to make the numeric literals more readable for you, but you cannot put more than one underline between each digit. The poor choice of underline locations in this example do not add to the ease of reading the numbers but illustrate the places where they can be located. The word **INTEGER** has been omitted from lines 12 and 13, which makes the type of these constants slightly different from the other two but we will have to learn a bit more before we can understand or appreciate the value of doing this.

Lines 17 through 25, in the executable part of the program, should be easy for you to understand on your own, so they will be left to your study.

DECLARING LITERAL CONSTANTS

Lines 27 through 29 give examples of declaration of literal values using the exponential notation. The exponent can be indicated with either case of "E" and the number following it must be positive for an **INTEGER** literal. Lines 30 through 33 give examples of the use of a base other than 10. The radix of the number is given prior to the first "#" sign, and can be any value from 2 through 16, the radix itself being given in decimal notation. The value is given between "#" signs and can be followed by an optional exponent, the exponent being given in the defined base. If no radix is given, base 10 is assumed as in lines 27 through 29 of this example program.

The executable part of this program should be very clear to you. Be sure to compile and execute it.

TYPES AND SUBTYPES

Example program ------> **e_c03_p4.ada**

```
                                        -- Chapter 3 - Program 4
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure SubTypes is

   type MY_INT_TYPE is range -10_000..20_000;
   My_Int : MY_INT_TYPE;

   package My_Int_IO is new Ada.Text_IO.Integer_IO(MY_INT_TYPE);
   use My_Int_IO;

   subtype SUB_INT is INTEGER range 12..144;
   Thing        : SUB_INT;
   Count        : INTEGER;
   Stuff        : INTEGER range 12..144;

   START        : constant := 4;                       -- START is 4
   STOP         : constant := START + 13;              -- STOP is 17
   Example      : SUB_INT range 4*START..2*STOP + 7;   -- 16..41
   Example2     : MY_INT_TYPE range START..2*STOP + 7; -- 4..41

begin

   Thing := 6 * 3;                          -- Thing is 18
   Count := Thing + 17;                     -- Count is 35
   Stuff := Count - Thing;                  -- Stuff is 17

   My_Int := 6 * 3;                         -- My_Int is 18
   My_Int := MY_INT_TYPE(Thing + 17);       -- My_Int is 35
   My_Int := MY_INT_TYPE(Thing) + 17;       -- My_Int is 35

   My_Int := MY_INT_TYPE(Thing) - 10;       -- My_Int is 8
   My_Int := MY_INT_TYPE(Thing - 10);       -- Run Time Error

   My_Int := 35;
   Thing  := SUB_INT(My_Int - 17);          -- Thing is 18
   Thing  := SUB_INT(My_Int - 27);          -- Run Time Error

   Example := Thing + SUB_INT(My_Int);

end SubTypes;




-- Result of execution

-- Exception never handled: constraint_error
-- Value 8 out of range 12..144.
```

Examine the program e_c03_p4.ada for your first look at a user defined type. We mentioned earlier that the range of a variable of type **INTEGER** could be different on different computers or with different compilers on the same computer. Ada gives us the ability to define our own type in such a way that it will be identical on every computer and with every Ada compiler.

A DIFFERENT TYPE IS INCOMPATIBLE

Line 7 defines a new integer type which will cover the range of -10,000 to 20,000 because of the use of the reserved word **range** to limit the available range of values that can be assigned to a variable of this type. Notice carefully that we called this an integer type, not a type **INTEGER**. Since it is an integer type, it has all of the properties defined earlier in this chapter but a variable of this type can only be assigned a value from -10,000 to 20,000. The actual range is given by specifying the lower and upper limits separated by two decimal points.

We have actually defined an entirely new type, and since Ada does very strong type checking, it will not allow you to assign a value directly from a variable of type **INTEGER** to a variable of our new type, or vice versa. The variable **My_Int** is defined with the new type in line 8. We will return to consideration of this variable later.

The package declaration in line 10 is completely new to us, so a few comments on it are in order. The complete definition of line 10 in Ada terminology will be given first, including a lot of new words which will mean very little to you until you gain some additional knowledge which will be given later in this tutorial. Line 10 is an instantiation of the generic package named **Ada.Text_IO.Integer_IO** with the type **MY_INT_TYPE** to provide the ability to output values of **MY_INT_TYPE** to the monitor or to files, or to input from the keyboard or from files. Saying the same thing in a more useful way, if you want to input or output values in some integral class type other than the predefined type **INTEGER**, you must include the line as shown with the desired type in the parentheses. Line 11 is included to make the new package easy to use. You will notice that it tells the system to **use** the package named in line 10 immediately following the keyword **package**.

As stated earlier, in an attempt to simplify the Ada learning curve, some topics will be deferred until later to give you a chance to absorb a few topics at a time, so don't spend any time trying to understand the previous paragraph at this time.

A NEW SUBTYPE IS COMPATIBLE WITH ITS PARENT

In line 13, we define a new subtype which is of the parent type **INTEGER** except that it covers a limited range, then we declare a variable named **Thing** of the new subtype in line 14. Any attempt to assign a value to the variable **Thing** which is outside of its assigned range, will result in an error. If you were running a small company with 23 employees and you assigned them each a unique number from 1 to 23, you would be interested if the payroll program tried to generate a paycheck for employee numbered 36, for example, because there would definitely be something wrong. A limited subrange could save you a lot of money in a case such as that.

ASSIGNMENT MUST BE TYPE COMPATIBLE

Anytime a value is assigned to a variable, the type assigned to it must be of its declared type or a compiler error will be generated. This will always be true in Ada and is one of the most important concepts behind its design. This is to prevent us from making the silly little mistakes which we humans are so good at making.

The variable **Count** is defined as an **INTEGER** type variable and **Stuff** is defined as a limited range **INTEGER** also. Remember that **Thing** is declared to be a **subtype** of **INTEGER**, so it is also a limited range **INTEGER** type variable. Because of the way these three variables are defined, they can be freely assigned to each other as long as the values assigned are within their respective ranges. This is because they are all of their parent type of **INTEGER**, and various assignments among them are illustrated in lines 25 through 27.

If we tried to assign the value of **Thing** to **My_Int**, the computer would give a compile error because they are of different types, but an explicit type conversion can be used to do the assignment as illustrated in line 30. By including the variable to convert to a new type in parentheses and preceding the parentheses with the desired type name, the system will convert the type as illustrated. The addition to **Thing** is completed and the entire expression inside of the parentheses is changed in type, by the explicit type conversion, to the desired type of the left side of the assignment statement. Note that explicit type conversion can only be done from within the same type class, in this case the integer class.

In line 31, the type is changed to the new type before the value of 17 is added to it, which should lead you to ask a question about the type of the constant 17.

HOW CAN 17 BE ADDED TO EITHER TYPE?

The constant 17 is of a very special type defined by Ada as type "universal_integer" which can be combined with any of the integer types without specific conversion. This term will be used in many ways in future lessons. The type universal_integer is compatible with all integer types and has a range with no limits. The range is effectively minus infinity to plus infinity. The type universal_integer is used for all literal values, but it is not available for your use in declaring a variable.

NOW FOR A SUBRANGE ERROR

Lines 30 and 31 are essentially the same because they result in the same answer, and lines 33 and 34 would appear to be the same, but they are not. In line 33, the value of **Thing**, which is 18, is converted to type **MY_INT_TYPE** and 10 is subtracted from it. Both 18 and 8 are within the specified range of **MY_INT_TYPE** so there is no error. In line 34 however, the result of the subtraction has the type of **Thing** and even the intermediate result is required to be within the range of 12 to 144. The result of 8 is outside of the required range so a run-time error is signaled in a special way called raising an exception. We will discuss the exception shortly but first notice that if we could get past the error, we could change the type of the result to **MY_INT_TYPE** and everything would work as desired.

INTERMEDIATE RESULTS OF CALCULATIONS

The ARM allows the limits of intermediate results to be checked against the limits of the parent type rather than the limits of the subtype. Line 34 therefore, may not give an error indicating that the intermediate result is out of the allowable range. This will depend on your compiler.

WHAT IS AN EXCEPTION?

We will have a lot to say about exceptions as we progress through this tutorial but a very brief description is needed at this time. When an Ada program is running, and a potentially disastrous error is detected, it would be good for you, the programmer, to be able to tell the system what to do with the error rather than cause a complete termination of the program. Ada gives you that capability through the use of exception handlers that you write and include in your program. In the above case, when the program detected the value of 8 as being out of the allowable range of that type of variable, it would signal your program that the error occurred, and if you did nothing, the Ada system would terminate the program. The proper Ada terminology for signaling you is called "raising an exception", and in the case of an out-of-bounds value, the exception named **Constraint_Error** would be raised. You could then trap this error and handle it any way you choose to.

There are several different exceptions that the system can raise and you can define your own exceptions which your program can raise and respond to. We will have a lot more to say about exceptions later in this tutorial.

A little study on your part should reveal why line 38 also has a run time error that will raise the exception **Constraint_Error**, if execution continues to that statement.

ANOTHER LOOK AT THE universal_integer

Before we leave this program we must look at lines 18 and 19 where we first define **START** and **STOP** as constants with no type indication. These constants are therefore of type "universal_integer" and can be used in the range definitions of the next two lines even though the two lines are of different parent types. If we had included the word **INTEGER** in the definitions in lines 18 and 19, they could not be used to define the limits of **Example2** because it is of a different type. The example program named e_c03_p3.ada had examples of **INTEGER** constants (**DOZEN** and **GROSS**) and universal_integer type constants (**BIG_NO** and **TWO**). The type universal_integer is actually a hidden type that is type compatible with all integer types.

It should also be observed that the limits of the range in these definitions are based on previously defined entities and can be of arbitrary complexity as long as they evaluate to the right types. The limits must also be within the range of the parent type or a compile error will result.

HOW TO DEFINE TYPES AND SUBTYPES

To declare types or subtypes, here are two simple formulas which can be followed.

```
   type    <type_name>    is <type_definition>;
   subtype <subtype_name> is <subtype_definition>;
```

Note that each declaration starts with a reserved word and includes the reserved word **is** between the name and the definition. It should be pointed out that Ada permits new types, subtypes, and variable declarations to be done in any order as long as everything is defined before it is used.

Compile and execute e_c03_p4.ada and observe the exception error as reported by your runtime system. The exception may be raised at line 34, depending on your compiler, where the value of 8 is outside of the allowable range of 12 through 144. The error message will vary with different compilers.

Much more will be said about types and subtypes in chapter 7 of this tutorial.

WHAT ARE ATTRIBUTES?

Example program ------> e_c03_p5.ada

```
                                       -- Chapter 3 - Program 5
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure IntAttrs is

   type BUG_RANGE is range -13..34;

   Rat : INTEGER;
   Dog : NATURAL;
   Cat : POSITIVE;
   Bug : BUG_RANGE;

begin

   Rat := 12;
   Dog := 23;
   Cat := 31;
   Bug := -11;

   Put("The type INTEGER uses ");
   Put(INTEGER'SIZE);
   Put(" bits of memory,");
   New_Line;
   Put(" and has a range from ");
   Put(INTEGER'FIRST);
   Put(" to ");
   Put(INTEGER'LAST);
   New_Line;
   Put(" Rat has a present value of ");
   Put(Rat);
   New_Line(2);

   Put("The type NATURAL uses ");
   Put(NATURAL'SIZE);
   Put(" bits of memory,");
   New_Line;
   Put(" and has a range from ");
   Rat := NATURAL'FIRST;
   Put(Rat);
   Put(" to ");
   Rat := NATURAL'LAST;
   Put(Rat);
   New_Line;
   Put(" Dog has a present value of ");
   Put(Dog);
   New_Line(2);

   Put("The type POSITIVE uses ");
   Put(POSITIVE'SIZE);
   Put(" bits of memory,");
   New_Line;
   Put(" and has a range from ");
   Put(POSITIVE'FIRST);
   Put(" to ");
   Put(POSITIVE'LAST);
   New_Line;
   Put(" Cat has a present value of ");
   Put(Cat);
   New_Line(2);

   Put("The type BUG_RANGE uses ");
   Put(INTEGER(BUG_RANGE'SIZE));
   Put(" bits of memory,");
   New_Line;
   Put(" and has a range from ");
   Put(INTEGER(BUG_RANGE'FIRST));
   Put(" to ");
   Put(INTEGER(BUG_RANGE'LAST));
   New_Line;
   Put(" Bug has a present value of ");
   Put(INTEGER(Bug));
   New_Line(2);

end IntAttrs;




-- Result of execution

-- The type INTEGER uses          32 bits of memory,
--  and has a range from -2147483648 to  2147483647
--  Rat has a present value of          12

-- The type NATURAL uses          31 bits of memory,
--  and has a range from           0 to  2147483647
--  Dog has a present value of          23

-- The type POSITIVE uses          31 bits of memory,
--  and has a range from           1 to  2147483647
--  Cat has a present value of          31

-- The type BUG_RANGE uses           7 bits of memory,
--  and has a range from         -13 to          34
--  Bug has a present value of         -11
```

Ada has a rather large list of attributes available for you as a programming aid. For an example of a program that contains, and therefore illustrates the use of attributes, examine the program named e_c03_p5.ada.

A new type is defined in line 7 with a limited range to illustrate that attributes are available even for user defined types.

Two additional predefined integer types are introduced in lines 10 and 11, the **NATURAL** and **POSITIVE** types. The **POSITIVE** type includes all integers greater than or equal to 1, and the **NATURAL** type includes all integers greater than or equal to 0. Both of these are available with your Ada compiler, and both are subtypes of **INTEGER**, so all variables of these three types can be freely mixed with no type errors.

Attributes are used to gain access to various limits within the program. For example, it may be necessary to know the upper limit of a variable of some type because it is of some strange subtype. An attribute can be used to find this limit, specifically the attribute **LAST** as illustrated in line 28 of this program. By combining the type name and the attribute in question with a "tick" or apostrophe, the upper limit of the range of the type is returned. The attribute **FIRST** is used to find the lowest value allowed by the subrange. The attribute **SIZE** gives the storage size of the type in bits of memory. Other attributes are available for integer types. A complete list of available attributes is given in Annex K of the ARM. You should spend a few minutes reviewing the list at this time even though you will understand little of what is presented there. In the near future, you will understand most of the material included there.

TYPE CONVERSIONS

Note the type conversions in lines 63, 67, 69, and 72. The values could be output directly with the **BUG_RANGE** type by instantiating a copy of **Ada.Text_IO.Integer_IO** in a manner similar to that done in the example program named e_c03_p4.ada, but we choose to use type conversion rather than instantiating a new copy of the I/O package. Because **NATURAL** and **POSITIVE** are subtypes of **INTEGER**, they use the copy of the I/O package pre-instantiated for the **INTEGER** type by the Ada system.

Compile and run this program and you will get a listing of the number of bits required by your compiler to store each of the four types along with the range covered by each of the four types.

PROGRAMMING EXERCISES

1. Write a program with some types containing some constrained limits and see what errors occur when you exceed their limits.(Solution)

-- Chapter 3 - Programming Exercise 1

```
procedure Ch03_1 is

   subtype RESTRICTED_RANGE is INTEGER range 12.. 77;

   Restricted_Variable : RESTRICTED_RANGE;

begin
   Restricted_Variable := 5;
   Restricted_Variable := 125;
   Restricted_Variable := 15 * (15 - 5);
end Ch03_1;




-- Result of Execution
--  (You will get an error for the executable statements, fix
--   each one and observe the error at the next step.
```

2. Try to assign some wrong type of data to some variables and try to mix up some types in arithmetic statements. Study the compiler error messages.(Solution)

-- Chapter 3 - Programming Exercise 2

```
procedure Ch03_2 is

   Index, Count : INTEGER;
   type NEW_INT is range 0..1200;
   New_Index, New_Count : NEW_INT;

begin
   Index := 100;
   New_Index := 100;
   Index := New_Index;
   Count := Index + New_Index;
   New_Count := Index + New_Index;
end Ch03_2;




-- Result of execution

--  (This has compilation errors in lines 12, 13, and 14.)
```

3. Modify e_c03_p5.ada to output the **BUG_RANGE** attributes directly by instantiating a new copy of the generic package **Ada.Text_IO.Integer_IO.** Don't spend too much time on this exercise before you look at the answer. It contains a new construct, at least new to you. (Solution)

```
                                    -- Chapter 3 - Programming Exercise 3
                                                 -- Chapter 3 - Program 5
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure Ch03_3 is

   type BUG_RANGE is range -13..34;

   package Bug_IO is new Ada.Text_IO.Integer_IO(BUG_RANGE);
   use Bug_IO;

   Rat : INTEGER;
   Dog : NATURAL;
   Cat : POSITIVE;
   Bug : BUG_RANGE;

begin

   Rat := 12;
   Dog := 23;
   Cat := 31;
   Bug := -11;
   Put("The type INTEGER uses ");
   Ada.Integer_Text_IO.Put(INTEGER'SIZE);
   Put(" bits of memory,");
   New_Line;
   Put(" and has a range from ");
   Put(INTEGER'FIRST);
   Put(" to ");
   Put(INTEGER'LAST);
   New_Line;
   Put(" Rat has a present value of ");
   Put(Rat);
   New_Line(2);

   Put("The type NATURAL uses ");
   Ada.Integer_Text_IO.Put(NATURAL'SIZE);
   Put(" bits of memory,");
   New_Line;
   Put(" and has a range from ");
   Rat := NATURAL'FIRST;
   Put(Rat);
   Put(" to ");
   Rat := NATURAL'LAST;
   Put(Rat);
   New_Line;
   Put(" Dog has a present value of ");
   Put(Dog);
   New_Line(2);

   Put("The type POSITIVE uses ");
   Ada.Integer_Text_IO.Put(POSITIVE'SIZE);
   Put(" bits of memory,");
   New_Line;
   Put(" and has a range from ");
   Put(POSITIVE'FIRST);
   Put(" to ");
   Put(POSITIVE'LAST);
   New_Line;
   Put(" Cat has a present value of ");
   Put(Cat);
   New_Line(2);

   Put("The type BUG_RANGE uses ");
   Bug_IO.Put(BUG_RANGE'SIZE);
   Put(" bits of memory,");
   New_Line;
   Put(" and has a range from ");
   Put(BUG_RANGE'FIRST);
   Put(" to ");
   Put(BUG_RANGE'LAST);
   New_Line;
   Put(" Bug has a present value of ");
   Put(Bug);
   New_Line(2);

end Ch03_3;




-- Result of execution
-- The type INTEGER uses     32 bits of memory
--  and has a range from -2147483648 to  2147483647
--  Rat has a present value of     12

-- The type NATURAL uses     31 bits of memory
--  and has a range from      0 to  2147483647
--  Dog has a present value of     23

-- The type POSITIVE uses     31 bits of memory
--  and has a range from      1 to  2147483647
--  Cat has a present value of     31

-- The type BUG_RANGE uses     7 bits of memory
--  and has a range from    -13 to     34
--  Bug has a present value of    -11
```

Ada Tutorial - Chapter 4

LOGICAL COMPARES AND PRECEDENCE

WHAT IS A BOOLEAN VARIABLE?

Example program ------> **e_c04_p1.ada**

```
                                        -- Chapter 4 - Program 1
with Ada.Text_IO;
use Ada.Text_IO;

procedure Compare is

   package Enum_IO is new Ada.Text_IO.Enumeration_IO(BOOLEAN);
   use Enum_IO;

   One   : INTEGER := 1;
   Two   : INTEGER := 2;
   Three : INTEGER := 3;

   Is_It : BOOLEAN := TRUE;      -- initialized
   Which : BOOLEAN;              -- uninitialized

begin

   Which := TRUE;
   Put("Which now has the value of ");
   Put(Which);
   New_Line;
   Which := FALSE;
   Put("Which now has the value of ");
   Put(Which);
   New_Line;

   Is_It := (One + 1) = Two;
   Is_It := One /= Two;
   Is_It := One + Two >= Three;

end Compare;




-- Result of execution

-- Which now has the value of TRUE
-- Which now has the value of FALSE
```

Examine the program named e_c04_p1.ada for an example of logical compares being used in a very trivial way. We declare and initialize three **INTEGER** type variables for use later then declare two variables of type **BOOLEAN** in lines 14 and 15, with the first one being initialized to **TRUE**. A **BOOLEAN** type variable has a very limited range of values which can be assigned to it, namely **TRUE** or **FALSE**, and there are no mathematical operations available for use on variables of type **BOOLEAN**.

Much more will be said about the **BOOLEAN** type variable later in this tutorial, but we need a basic understanding of a boolean variable in order to study program control in the next chapter.

Lines 19 and 23 illustrate how you can assign a value of either **TRUE** or **FALSE** to a **BOOLEAN** variable. These illustrate literal assignment in much the same way that a literal value can be assigned to an **INTEGER** type variable. Because we wish to display **BOOLEAN** values, we instantiate a copy of the generic package **Enumeration_IO** for the **BOOLEAN** type. Once again, we will study the details of this later in the tutorial. This package makes it possible to output **BOOLEAN** values in lines 21 and 25.

BOOLEAN ASSIGNMENT STATEMENTS

Lines 28 through 30 are a bit more interesting because they illustrate assigning a calculated **BOOLEAN** value to a **BOOLEAN** variable. In line 28, the value of the **INTEGER** variable **One** has the literal 1 added to it and the total is compared to the value contained in the variable **Two**. The expression reduces to the expression 1 + 1 = 2, and since 1 + 1 is equal to 2, the expression evaluates to **TRUE** which is assigned to the **BOOLEAN** variable **Is_It**. The various steps in evaluation are illustrated below for the student with little or no experience using **BOOLEAN** expressions in some other programming language.

```
    Is_It := (One + 1) = Two;
    Is_It := (1 + 1) = 2;
    Is_It := 2 = 2;
    Is_It := TRUE;
```

The single equal sign is the **BOOLEAN** operator for equality, and if the two expressions being evaluated are of the same value, a value of **TRUE** will result. If they are not of the same value, the **BOOLEAN** result will be **FALSE**.

ARE THERE OTHER BOOLEAN OPERATORS?

There are six **BOOLEAN** operators, and all six will be illustrated in the next example program. Line 29 in the present program illustrates use of one of the others, the inequality operator. This statement says, if **One** is not equal to **Two** then assign the value of **TRUE** to the **BOOLEAN** variable **Is_It**, otherwise assign a value of **FALSE** to the **BOOLEAN** variable **Is_It**. Note that regardless of the result, a value will be assigned to the **BOOLEAN** variable. Finally, the "greater than or equal" operator is illustrated in use in line 30.

This is a rather silly program since none of the results are used, but it is meant to illustrate just what a **BOOLEAN** variable is and how to use it. Compile and run this program before continuing on to the next example program.

ADDITIONAL TOPICS ON BOOLEAN EVALUATION

Example program ------> **e_c04_p2.ada**

```
                                       -- Chapter 4 - Program 2
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure Compares is

   package Enum_IO is new Ada.Text_IO.Enumeration_IO(BOOLEAN);
   use Enum_IO;

   Index, Count           : INTEGER := 12;
   Truth, Lies, Question  : BOOLEAN;

begin
   Truth := Index = Count;      -- This is TRUE
   Lies := Index < Index;       -- This is FALSE

                            -- Examples of all BOOLEAN operators
   Question := Index =  Count;      -- Equality
   Question := Index /= Count;      -- Inequality
   Question := Index <  Count;      -- Less than
   Question := Index <= Count;      -- Less than or equal
   Question := Index >  Count;      -- Greater than
   Question := Index >= Count;      -- Greater than or equal

                       -- Examples of composite BOOLEAN expressions
   Question := Index = 12 and Count = 12 and Truth and TRUE;
   Question := Index /= 12 or FALSE or Count > 3 or Truth;
   Question := (Truth or Lies) and (Truth and not Lies);
   Question := Truth xor Lies;

                       -- now for short circuit evaluation
   Question := Index /= Count and then Index = 9/(Index - Count);
   Question := Index = Count or else  Index = 9/(Index - Count);
   Question := (Index = Count) or else (Index = 9/(Index - Count));

end Compares;




-- Result of execution

--   (No output generated by this program.)
```

Examine the program named e_c04_p2.ada and you will see all six **BOOLEAN** operators in use. We define two **INTEGER** and three **BOOLEAN** type variables in the declaration part of the program, and begin the executable part with some compare examples in lines 14 and 15. It should be clear that in line 15, **Index** can not be less than itself, so the result is **FALSE**.

Lines 18 through 23 illustrate the use of all six **BOOLEAN** operators which are available in Ada and you should have no trouble understanding this list.

THE BOOLEAN and OPERATOR

Lines 26 through 29 illustrate composite **BOOLEAN** operators using the reserved words **and**, **or**, **not**, and **xor**. The statement in line 26 says that

```
        if Index = 12
    and if Count = 12
    and if Truth currently has the value TRUE
    and if TRUE     (which is always TRUE)
       then assign TRUE to Question
  otherwise assign FALSE to Question.
```

Using the freeform available in an Ada program to write the previous sentence, can help in understanding the sentence. The point to be illustrated is that you can combine as many **BOOLEAN** expressions as you desire to do a particular job.

THE BOOLEAN or OPERATOR

Using the expanded freeform available in Ada, the statement in line 27 says that

```
        if Index is not equal to 12
     or if FALSE     (which is always FALSE)
     or if Count is greater than 3
     or if Truth currently has the value of TRUE
       then assign TRUE to Question
  otherwise assign FALSE to Question.
```

THE BOOLEAN not OPERATOR

The expression in line 28 illustrates the use of these two operators combined in a slightly more complex way using parentheses to properly group the expressions. A new operator appears here, the **not** which simply says to reverse the meaning of the **BOOLEAN** operator which it precedes.

THE BOOLEAN xor OPERATOR

Line 29 illustrates the use of the "exclusive or" operator which says that the result will be **TRUE** if one and only one of the operands are **TRUE**, and **FALSE** if both operands are **TRUE**, or both operands are **FALSE**. A good illustration of this operation would be a hall light with a switch at both ends of the hall. If one of the switches is up, the light is on, but if both are up or both are down, the light is off.

FULL EVALUATION OF BOOLEAN EXPRESSIONS

The way the expressions are written in lines 26 through 29, all of the expressions will be evaluated when the statement is executed. If, in the case of line 26, the value of **Index** is not 12, then the final result will be **FALSE** no matter what the rest of the expressions are and it would be a waste of time to evaluate them. Ada will continue blindly across the entire line evaluating all of the expressions and wasting time since it should know the final result based on the first comparison. There is however, a way to tell it to stop as soon as it knows the final answer, through use of the short circuit operators.

WHAT ARE "SHORT CIRCUIT OPERATORS"?

If you study line 32, you will see that if **Index** is equal to **Count**, we will be dividing the constant 9 by zero in the second part of the expression. By adding the reserved word **then** to the **and** operator we have a short circuit operation, which means as soon as the system knows the final outcome, the remaining operations are short circuited and not evaluated. In the present example, if **Index** is equal to **Count**, the first term is **FALSE** and there is no need to continue since the second term is to be **and**ed with the **FALSE** resulting in **FALSE** no matter what the second term is. Division by zero is avoided in this case because the division is not attempted. In the same manner, line 33 illustrates the short circuit **or** operator which is obtained by adding the reserved word **else**. In this case, if **Index** is equal to **Count**, the result will be **TRUE** regardless of what the second term is, so the second term is not evaluated and division by zero is avoided. Line 34 is identical to line 33 but illustrates the use of parentheses to make the logic a little easier to read.

It should be clear that Ada provides the tools needed to do any boolean operation needed. It is up to you to learn how to use them.

ORDER OF PRECEDENCE

The order of precedence of operators is given by the following list.

```
     **     not    abs      -- Highest precedence

     *      /      mod rem  -- Multiplying operators

     +      -               -- Unary operators

     +      -      &        -- Binary adding operators

     =      /=     <        -- Relational operators
     <=     >      >=       -- Relational operators
     in     not in          -- (same precedence)

     and    or     xor      -- Logical operators
     and then      or else  -- (same precedence)
```

The Ada 95 Reference Manual (ARM) has a complete list of the operators and the details of the order of precedence in section 4.5. If there is any question as to the order of precedence, you should group expressions together with parentheses since they have the absolute highest precedence. A future reader of your program will know exactly what your program is doing.

Be sure to compile and execute this program. Note that we have not yet studied the &, the **in**, and the **not in** operators but will soon.

PROGRAMMING EXERCISE

1. Add some output statements to both example programs to see that the results are as predicted. This will give you experience using the boolean output statement.(Solution)

```
                           -- Chapter 4 - Programming exercise 1
                                        -- Chapter 4 - Program 1
with Ada.Text_IO;
use Ada.Text_IO;

procedure Ch04_1 is

   package Enum_IO is new Ada.Text_IO.Enumeration_IO(BOOLEAN);
   use Enum_IO;

   One   : INTEGER := 1;
   Two   : INTEGER := 2;
   Three : INTEGER := 3;

   Is_It : BOOLEAN := TRUE;      -- initialized
   Which : BOOLEAN;              -- uninitialized

begin

   Which := TRUE;
   Put("Which now has the value of ");
   Put(Which);
   New_Line;
   Which := FALSE;
   Put("Which now has the value of ");
   Put(Which);
   New_Line;

   Is_It := (One + 1) = Two;
   Put("Is_It now has the value of ");
   Put(Is_It);
   New_Line;
   Is_It := One /= Two;
   Put("Is_It now has the value of ");
   Put(Is_It);
   New_Line;
   Is_It := One + Two >= Three;
   Put("Is_It now has the value of ");
   Put(Is_It);
   New_Line;

end Ch04_1;




-- Result of execution

-- Which now has the value of TRUE
-- Which now has the value of FALSE
-- Is_It now has the value of TRUE
-- Is_It now has the value of TRUE
-- Is_It now has the value of TRUE
```

Ada Tutorial - Chapter 5

CONTROL STRUCTURES

THE SIMPLEST LOOP IN ADA

Example program ------> **e_c05_p1.ada**

```
                                       -- Chapter 5 - Program 1
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure LoopDemo is

   Index, Count : INTEGER;

begin
   Index := 1;
   loop                            -- This is the simplest loop
      Put("Index =");
      Put(Index, 5); New_Line;
      Index := Index + 1;
      exit when Index = 5;
   end loop;

   Index := 1;
   loop                            -- Another simplest loop
      Put("Index =");
      Put(Index, 5); New_Line;
      Index := Index + 1;
      if Index = 5 then exit; end if;
   end loop;

   Count := 1;
   while Count < 5 loop            -- This is the while loop
      Put("Count =");
      Put(Count, 5); New_Line;
      Count := Count + 1;
   end loop;

   for Index in 1..4 loop          -- This is the for loop
      Put("Doubled index =");
      Put(2 * Index, 5); New_Line;
   end loop;

   for Count in reverse 5..8 loop  -- This is the reverse for loop
      Put("Triple count =");
      Put(3 * Count, 5); New_Line;
   end loop;

   for Index in 7..11 loop         -- An empty loop
      null;
   end loop;

end LoopDemo;




-- Result of execution

-- Index =    1
-- Index =    2
-- Index =    3
-- Index =    4
-- Index =    1
-- Index =    2
-- Index =    3
-- Index =    4
-- Count =    1
-- Count =    2
-- Count =    3
-- Count =    4
-- Doubled index =    2
-- Doubled index =    4
-- Doubled index =    6
-- Doubled index =    8
-- Triple count =   24
-- Triple count =   21
-- Triple count =   18
-- Triple count =   15
```

We will start with the simplest loop in Ada, the infinite loop which is illustrated in lines 11 through 16 of the program named e_c05_p1.ada. The variable named **Index** is initialized to the value of 1 prior to entering the loop, and the loop itself is given in lines 11 through 16. The loop begins with the reserved word **loop** and ends with the two reserved words **end loop.** Any number of executable statements are placed between these two delimiters and the loop is repeated continuously until something causes the program to jump out of the loop. In this case, the variable **Index** is incremented each time through the loop, and when it reaches a value of 5, the **exit** statement is executed which causes control to jump out of the loop and begin executing instructions immediately following the end of the loop. The words **exit** and **when** are two more reserved words.

The expression following the **exit when** reserved words must evaluate to a **BOOLEAN** result and when the result is **TRUE**, the loop is exited, but as long as the result is **FALSE**, the loop execution will continue. Note that the **exit** statement can be placed anywhere in the loop and as many conditional exits as needed can be placed within the loop.

The statements illustrated in lines 18 through 24 use an alternative form of loop exit which uses the **if** statement which we have not yet studied. The **if** form of exit is such a common form of usage that it had to be included here as one of the simplest types of loops. Note that the **if exit** can be placed anywhere in the loop and as many as needed can be used within the loop. The **if** statement will be fully explained later in this chapter.

THE exit STATEMENT

There is a subtle difference between the **exit** in line 15 and the **exit** in line 23. The exit in line 15 is conditional because it is only executed if the condition evaluates to **TRUE**. The exit in line 23 however, is unconditional since the exit will be taken anytime control reaches the word **exit**. Keep in mind that an **exit** is used only to exit a loop, it is not used for any other construct in Ada.

THE while LOOP

The **while** loop is identical to the simple loop except for the addition of a test prior to the reserved word **loop**. The test is done at the beginning of the loop so it is slightly less general than the simple loop, but it also requires a **BOOLEAN** expression as part of the construct. This loop is illustrated in lines 27 to 31 of the present program. In line 27, "while Count < 5" is called the iteration clause.

THE for LOOP

The loops studied so far in this example program use an indeterminate number of passes since they calculate their own limits as they progress through the loop. It is important to point out that the **BOOLEAN** expressions are evaluated on every pass through the loop. The **for** loop, however, has its limits evaluated one time and the number of passes through the loop is completely defined before the first pass through the loop is begun.

The **for** loop is illustrated in lines 33 through 36 where the control consists of a few words prior to the reserved word **loop**. In this case, the loop control is the variable **Index** and the range of the variable is 1 through 4. The loop will be executed four times, each time with a larger number for the variable **Index**, since the loop index is always incremented by one. There is no provision for an increment of any other value in Ada. The reserved words **for** and **in** are used in the manner shown for all **for** loops. They serve to bracket the loop index, in this case named **Index**, and the range of the index, in this case 1 through 4 inclusive.

Because the type **INTEGER** is so commonly used for the loop index, if the type is not specifically stated, it will be defaulted to the type **INTEGER**. This is to make it easier to code the majority of loops.

Note that the loop parameters do not need to be of type **INTEGER**, but the loop index and both range limits must be of the same type. Note also that the value of the loop iterator is not available after loop termination. The loop iterator ceases to exist completely when the loop is completed. (There is good reason for this as we will see when we complete our study of the next example program.)

THE BACKWARDS for LOOP

There is a slight variation to the **for** loop illustrated in lines 38 through 41, where the loop index is decremented on each pass through the logic. The word **reverse** is another reserved word and serves to indicate the backward counting nature of this loop. Note that the range is expressed in ascending order, but the actual execution begins at 8 and decrements by 1 on each pass until it reaches 5, where it quits after the pass through the loop with **Count** set to 5.

THE EMPTY LOOP

Continuing in the program named e_c05_p1.ada, the loop in lines 43 to 45 is given to illustrate that it is possible to write a loop that does absolutely nothing. It may seem like a silly thing to do, but there are cases, when using tasking, that it is necessary to do nothing in a loop. It is of more importance at this point to illustrate that Ada is so picky, you are not allowed to write an empty loop, but are required to inform the compiler that you really did mean for the loop to do nothing by including the reserved word **null** as a statement within the loop.

Be sure to compile and execute this program and be sure you understand exactly what each loop does.

SLIGHTLY MORE COMPLEX for STATEMENTS

Example program ------> **e_c05_p2.ada**

```
                                       -- Chapter 5 - Program 2
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure MoreLoop is

   type MY_TYPE is range 10..13;

   package My_Int_IO is new Ada.Text_IO.Integer_IO(MY_TYPE);
   use My_Int_IO;

   My_Range      : MY_TYPE;
   TWO           : constant INTEGER := 2;
   THREE         : constant INTEGER := 3;
   FOUR          : constant INTEGER := 4;
   Height,Width  : INTEGER;
   Special_Index : INTEGER;

begin

   for Index in MY_TYPE loop
      Put("Going through the first loop");
      Put(Index, 3);
      New_Line;
   end loop;

   for Index in MY_TYPE'FIRST..MY_TYPE'LAST loop
      Put("Going through the second loop");
      Put(Index, 3);
      New_Line;
   end loop;

   for Index in TWO..THREE**2 - FOUR loop    -- range is 2..5
      Put("Going through the third loop");
      Put(Index, 3);
      New_Line;
   end loop;

Named_Loop:
   for Height in TWO..FOUR loop
      for Width in THREE..5 loop
         if Height * Width = 12 then
            exit Named_Loop;
         end if;
         Put("Now we are in the nested loop and area is");
         Put(Height*Width, 5);
         New_Line;
      end loop;
   end loop Named_Loop;

   Special_Index := 157;
   for Special_Index in 3..6 loop
      Put("In the Special Index loop");
      Put(Special_Index, 5);
      New_Line;
   end loop;
   Put("The Special Index loop is completed");
   Put(Special_Index, 5);
   New_Line;

end MoreLoop;




-- Result of execution

-- Going through the first loop 10
-- Going through the first loop 11
-- Going through the first loop 12
-- Going through the first loop 13
-- Going through the second loop 10
-- Going through the second loop 11
-- Going through the second loop 12
-- Going through the second loop 13
-- Going through the third loop  2
-- Going through the third loop  3
-- Going through the third loop  4
-- Going through the third loop  5
-- Now we are in the nested loop and area is    6
-- Now we are in the nested loop and area is    8
-- Now we are in the nested loop and area is   10
-- Now we are in the nested loop and area is    9
-- In the Special Index loop    3
-- In the Special Index loop    4
-- In the Special Index loop    5
-- In the Special Index loop    6
-- The Special Index loop is completed  157
```

Examine the program named e_c05_p2.ada for a few examples of more complex but much more versatile **for** loops. The loops illustrate a few of the flexibilities designed into Ada to allow for more efficient programming. The first example in lines 21 through 25, has two new concepts, the first being that the loop variable, **Index**, is not declared in the definition part of the program, and secondly the range is defined by a type rather than variable or constant limits.

Considering the first point, the loop index in a **for** loop does not require explicit declaration, but will be implicitly declared by the program, used during the duration of the loop, and discarded when the loop terminates. The final value of the loop index is not available for use after the loop terminates, and this is true regardless of whether the index is explicitly or implicitly declared. (We will see shortly that it is always implicitly declared by the Ada system.) The Ada language designers gave compiler writers freedom on how and when the loop index is incremented rather than dictating what the final value would be. It is a simple rule to remember that you can not depend on having the final value of the loop index when you terminate the loop. If you need the final value, you must copy it into some other variable prior to leaving the loop.

USING A TYPE FOR THE LOOP RANGE

The second point brought out above is the fact that the loop range is given as a type. The type given, **MY_TYPE**, has a defined range of 10..13, so it should be no real problem seeing what the loop index range is. Moreover the implicit loop index, named **Index**, will also be of type **MY_TYPE**. Because we wish to print the value of the loop index during each pass through the loop, we instantiate a copy of the **Integer_IO** package for the **MY_TYPE** type in lines 9 and 10.

A final point about the first loop in this program must be emphasized. The loop index in this **for** loop, as well as in any **for** loop, will be treated as a constant within the loop, so you cannot assign a new value to it. The looping mechanism itself will be the only way that the loop index can be changed in value.

OUR FIRST ATTRIBUTES IN USE

In chapter 3, we took a brief look at attributes, but didn't really look at uses for them. In the loop in lines 27 through 31, we have the range once again defined as the limits of the type named **MY_TYPE** but this time we explicitly name the first and last values of the type by using the attributes. The method depicted in line 21 is more concise, but both methods lead to the same result. If you wanted to loop from the first element to some midpoint, for example, the second method gives you a way to use one of the endpoints of the range.

CALCULATED LOOP RANGE LIMITS

The loop in lines 33 through 37 has range limits that are not static but that are calculated when the loop is entered. In this case, the calculations are all based on constants, but they could be based on variables of any arbitrary degree of complexity. If they are based on one or more variables, there is a subtle point that you must understand. If one or more of the variables are changed within the loop, the range does not change accordingly, because the loop range limits are calculated only once, when the loop is entered.

It should be clear that the loop range is 2 through 5 for this particular loop.

A DOUBLY NESTED for LOOP

Lines 39 through 49 contain two nested **for** loops with an addition to the outer loop. The identifier **Named_Loop** is a loop name which applies to the outer loop. The name appears prior to the start of the loop and also follows the corresponding **end loop** for that loop. This results in a named loop. In lines 42 and 43 we have a conditional statement that says if the product of the **Height** and the **Width** are equal to 12, we are to exit the loop that is named **Named_Loop**. Without the name given we would only exit the loop currently in effect, the inner one defined by the loop index **Width**, but since we mention the name, we exit the loop with the given name, and exit both loops. By adding a name to a loop, you can exit out of as many nested levels as you desire with a single **exit** statement.

A CLOSE LOOK AT THE LOOP INDEX

In line 51, we assign the value of 157 to the variable named **Special_Index** and use that variable name for the loop index in the following loop. After the loop is terminated, we display the value of the variable to see what the final value is, and when we run the program we find that the value did not change from that assigned, namely 157. This would suggest that even if we think we are explicitly defining the loop index variable, the system is actually making up an entirely new variable, using it, and throwing it away after the loop is terminated. Ada defines a loop variable as an automatic variable which is generated when needed and discarded when it is no longer needed. More will be said about automatic variables later. The important point to grasp from this is that Ada implicitly declares all loop variables for us.

LOOPS ARE IMPORTANT

You will have many opportunities to use all three forms of the loops discussed here, so it would pay you to study these loops until you are sure of what they are doing.

Note the similarity of the three loops, and the fact that each is a complete Ada statement terminated with a semicolon. When we study the next two controls, you will see that they are very similar, and fit the same pattern.

```
                               loop <...> end loop;
   while <BOOLEAN expression>  loop <...> end loop;
   for <loop index> in <range> loop <...> end loop;
```

Compile and execute this program taking care to observe the value of the variable named **Special_Index** when the loop is complete.

This would be a good point for you to begin getting acquainted with the Ada 95 Reference Manual (ARM). Study section 5.5 in the ARM which defines the Loop Statements in detail. You will find it surprisingly easy to read and understand except for the syntax statements at the beginning of the section. You will see that there is little to the loop construct that we have not covered in this section of the tutorial, and you will begin getting familiar with the ARM. You will find the ARM to be a valuable source of information after you gain some experience with Ada.

NOW FOR CONDITIONAL STATEMENTS

Example program ------> e_c05_p3.ada

```
                                       -- Chapter 5 - Program 3
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure IfDemo is

begin
   for Index in 1..7 loop  -- This contains two simple if statements
      Put("Index is ");
      Put(Index, 3);
      if Index < 4 then
         Put(" so is less than 4");
      end if;
      if Index > 5 then
         Put(" so is more than 5");
      end if;
      New_Line;
   end loop;
   New_Line;

   for Index in 13..17 loop  -- This contains an else clause
      Put("Index is");
      Put(Index, 3);
      if Index < 15 then
         Put_Line(" and is less than 15.");
      else
         Put_Line(" and is 15 or greater.");
      end if;
   end loop;
   New_Line;

   for Index in 13..17 loop  -- This introduces the elsif statement
      Put("Index is");
      Put(Index, 3);
      if Index < 15 then
         Put_Line(" and is less than 15.");
      elsif Index = 15 then
         Put_Line(" and is 15.");
      elsif Index = 16 then
         Put_Line(" and is 16.");
      else
         Put_Line(" and is greater than 16.");
      end if;
   end loop;
   New_Line;

-- This final group of statements contains a loop with a nested if
--   statement, and a loop within the the else part of the nested
--   if statement.

   for Index in 13..17 loop
      Put("Index is");
      Put(Index, 3);
      if Index < 16 then
         if Index > 14 then
            Put(" and is less than 16 and greater than 14.");
         else
            Put(" and is less than or equal to 14.");
         end if;
      else
         Put(" and is 16 or greater.");
         for New_Index in 222..224 loop
           Put(" stutter");
         end loop;
      end if;
      New_Line;
   end loop;

end IfDemo;
```

```
-- Result of execution

-- Index is   1 so is less than 4
-- Index is   2 so is less than 4
-- Index is   3 so is less than 4
-- Index is   4
-- Index is   5
-- Index is   6 so is more than 5
-- Index is   7 so is more than 5
--
-- Index is 13 and is less than 15.
-- Index is 14 and is less than 15.
-- Index is 15 and is 15 or greater.
-- Index is 16 and is 15 or greater.
-- Index is 17 and is 15 or greater.
--
-- Index is 13 and is less than 15.
-- Index is 14 and is less than 15.
-- Index is 15 and is 15.
-- Index is 16 and is 16.
-- Index is 17 and is greater than 16.
--
-- Index is 13 and is less than or equal to 14.
-- Index is 14 and is less than or equal to 14.
-- Index is 15 and is less than 16 and greater than 14.
-- Index is 16 and is 16 or greater. stutter stutter stutter
-- Index is 17 and is 16 or greater. stutter stutter stutter
```

We will spend some time now examining the program named e_c05_p3.ada and studying the conditional statements. Notice here, that we don't even try to define any variables, but we will let the system generate them for us as implicit loop indices. The Ada statement contained in lines 8 through 18 is a **for** loop which we are now familiar with, so we will use it to study how the **if** statement works. Notice that we display the value of **Index**, then use it in the first **if** statement in lines 11 through 13. This statement says that if the value stored in **Index** is less than 4, then execute the statements between the reserved words **then** and **end if**. In this case, we display the message in line 12 on the monitor. If the value stored in **Index** is not less than 4, we ignore the statement between the reserved words and go directly to line 14, where we encounter another **if** statement. This one displays a different line on the monitor if the value stored in **Index** is greater than 5. This is the simplest kind of **if** statement, and you will find it to be extremely useful, in fact, you used it in the previous two example programs. Note that the **if** condition must evaluate to a **BOOLEAN** value.

Moving ahead to the next loop, the one in lines 21 through 29, we find a more elaborate conditional, the **if then else** statement. In this case, if the value of **Index** is less than 15, one message is displayed and if not, a different message is displayed. In the prior loop, we used an **if** statement where a message may or may not have been displayed, but in this one, we use an **if** statement where one of the messages will be displayed every time. This, of course, is very similar to other programming languages.

NOW FOR THE MULTIWAY if STATEMENT

The loop in the statement contained in lines 32 through 44 is a bit more involved since there are several possibilities for selection of the line to be displayed. The word **elsif** is another reserved word. In this case, one and only one of the statements will be executed during each pass through the loop. The reserved word **else** is optional, and if it were not there, it would be possible that none of the conditions would be met and therefore none of the messages would be output. As many statements as desired can be placed between each of the reserved words such that a selection could cause a large number of statements to be executed sequentially.

The last statement, given in lines 51 through 67, gives an illustration of nesting of **loop** and **if** statements. It should cause you no difficulty to understand the nesting and when you compile and run this program, you should have a good understanding of the output.

THE MULTIWAY CONDITIONAL CONSTRUCT

Example program ------> **e_c05_p4.ada**

```
                                       -- Chapter 5 - Program 4
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure CaseDemo is

begin
   for How_Many in 4..13 loop
      Put("We now have ");
      Put(How_Many, 3);
      Put(" widgets, ");
      case How_Many is
         when 4..6    => Put("which is too few.");
         when 7|9     => Put("but we don't need 7 or 9.");
         when 13      => Put("but that is too many.");
         when 8|10|12 => Put("which is a large even number.");
         when 11      => Put("enough for a football team.");
      end case;
      New_Line;
   end loop;
   New_Line;

   for How_Many in 100..105 loop
      Put("It is now ");
      Put(How_Many, 3);
      Put(" ");
      case How_Many is
         when 100 => Put("The value is 100, and useless.");
         when 101 => for Index in 2..5 loop
                        Put("Puppy ");
                     end loop;
         when 103 => if TRUE then
                        Put("Of course TRUE will always be true.");
                     end if;
         when 105 => null;
         when others => Put("This is one of those not defined.");
      end case;
      New_Line;
   end loop;
end CaseDemo;




-- Result of execution

-- We now have   4 widgets, which is too few.
-- We now have   5 widgets, which is too few.
-- We now have   6 widgets, which is too few.
-- We now have   7 widgets, but we don't need 7 or 9.
-- We now have   8 widgets, which is a large even number.
-- We now have   9 widgets, but we don't need 7 or 9.
-- We now have  10 widgets, which is a large even number.
-- We now have  11 widgets, enough for a football team.
-- We now have  12 widgets, which is a large even number.
-- We now have  13 widgets, but that is too many.
--
-- It is now 100 The value is 100, and useless.
-- It is now 101 Puppy Puppy Puppy Puppy
-- It is now 102 This is one of those not defined.
-- It is now 103 Of course TRUE will always be true.
-- It is now 104 This is one of those not defined.
-- It is now 105
```

Examine the program named e_c05_p4.ada for an example of a program with a case statement, the multiway conditional statement in Ada. Lines 8 through 20 comprise a loop with a **case** statement contained in it. Lines 12 through 18 contain the actual case statement. The variable **How_Many** is the case selector variable and each time we pass through the loop, we enter the case statement with a different value stored in the variable **How_Many**. When **How_Many** has the value of 4 through 6, we select the first path, and that path only, when it has the value of 7 or 9, we select the second path, etc. The vertical bar is the **or** operator here, and means if we have either of these cases, do this list of statements. The case statement is composed of the reserved words **case**, **is**, **when**, and **end case** in the manner shown. All cases must be accounted for in some way since it is an error to enter the case with a value of the case selector variable for which there is not something defined for it to do.

WHAT IS THE => OPERATOR?

The "=>" operator is used in many places in an Ada program which we will see as we progress through this tutorial. It can be loosely defined as a "do this" operator, because in most places it can be read as "do this". For example in line 13, the line can be read "when **How_Many** is 4 through 6 do this **Put** statement".

A special case is provided which is the **others** case. It is optional, but if it is included, it must be the last one listed. This is illustrated in the other example in this program, the example in lines 27 through 37. This program is an illustration of more complex operations in each of the case branches, and it is expected that you should have no difficulty discerning the operation of each of the cases. This is one of the places where a **null** statement can be very useful as in line 35 where we really want to do nothing and can explicitly tell the compiler so. Be sure to compile and execute this program after you are sure you understand it. The **case** statement occurs rather infrequently in programming, but you will use it, so you should understand it thoroughly.

This would be a good time to repeat a list which was given earlier in this chapter with the **if** and **case** statements added to illustrate the symmetry of the control structures and how well thought out they really are.

```
                              loop <...> end loop;
   while <BOOLEAN expression>  loop <...> end loop;
   for <loop index> in <range> loop <...> end loop;
   if <condition>              then <...> end if;
   case <selector>             is   <...> end case;
```

THE EVIL GOTO STATEMENT

Example program ------> e_c05_p5.ada

```
                                                -- Chapter 5 - Program 5
with Ada.Text_IO;
use Ada.Text_IO;
```

```
procedure GoToDemo is

begin
   goto Some_Place;

<<There>>
   Put_Line("I am now at the There place.");
   goto Stop_This_Mess;

<<Where>>
   Put_Line("I am now at the Where place.");
   goto There;

<<Some_Place>>
   Put_Line("I am now Some_Place.");
   goto Where;

<<Stop_This_Mess>>
   Put_Line("I am now about to end this mess.");

end GoToDemo;




-- Result of execution

-- I am now Some_Place.
-- I am now at the Where place.
-- I am now at the There place.
-- I am now about to end this mess.
```

The program named e_c05_p5.ada illustrates use of the **goto** statement, a statement that has fallen into ill repute among software developers in recent years. This is because the **goto** can be used very abusively, leading to nearly unreadable spaghetti code. There may be instances when use of the **goto** statement will result in very clear code, and when those cases arise, it should be used. It has been proven that it is possible to write any logic without resorting to the **goto** statement by using only iteration (looping), conditional branching, and sequential statements and every effort should be made to rely on these three elements alone.

When Ada was being developed, the developers considered leaving the **goto** statement completely out of the language, but it was included for a very nebulous reason. It was felt that after Ada became a popular language, there would be a need to translate existing programs from older languages, such as FORTRAN, into Ada using automatic translators. Such a job would be impossible without the **goto** statement, so it was included in the Ada language. Modula-2 is an example of a language structured very much like Ada, but without a **goto** statement defined as part of the language.

The **goto** is available as a part of the Ada language and is illustrated in the present example program. It should be no surprise to you that **goto** is another reserved word. The rather messy looking double "<<" and ">>" enclose the label to which you wish to jump and the rest should be easy for you to discern. Be sure to compile and run this program.

FINALLY, A USEFUL PROGRAM

Example program ------> **e_c05_p6.ada**

```
                                     -- Chapter 5 - Program 6
```

```
-- Centigrade to Farenheit temperature table
--
--  This program generates a list of Centigrade and Farenheit
--    temperatures with a note at the freezing point of water
--    and another note at the boiling point of water.

with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure TempConv is

   Centigrade, Farenheit : INTEGER;

begin
   Put("Centigrade to Farenheit temperature table");
   New_Line(2);
   for Count in INTEGER range -2..12 loop
      Centigrade := 10 * Count;
      Farenheit := 32 + Centigrade * 9 / 5;
      Put("C =");
      Put(Centigrade, 5);
      Put("    F =");
      Put(Farenheit, 5);
      if Centigrade = 0 then
         Put("  Freezing point of water");
      end if;
      if Centigrade = 100 then
         Put("  Boiling point of water");
      end if;
      New_Line;
   end loop;
end TempConv;




-- Result of execution

-- Centigrade to Farenheit temperature table
--
-- C =  -20    F =   -4
-- C =  -10    F =   14
-- C =    0    F =   32  Freezing point of water
-- C =   10    F =   50
-- C =   20    F =   68
-- C =   30    F =   86
-- C =   40    F =  104
-- C =   50    F =  122
-- C =   60    F =  140
-- C =   70    F =  158
-- C =   80    F =  176
-- C =   90    F =  194
-- C =  100    F =  212  Boiling point of water
-- C =  110    F =  230
-- C =  120    F =  248
```

Examine the program named e_c05_p6.ada for an example of a program that really does do something useful, it generates a table of Centigrade to Fahrenheit temperature conversions, and it outputs a special message at the freezing point of water and another at the boiling point of water. Notice the program definition in the header that defines exactly what it does. You should have no trouble understanding every detail of this program, except of course for the rather cryptic code in lines 8 and 9. Although we have had a brief introduction to these, we will cover the details of them later in this tutorial.

POOR CHOICE OF VARIABLE NAMES

Example program ------> **e_c05_p7.ada**

```
                                       -- Chapter 5 - Program 7
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure DumbConv is

   X, Y : INTEGER;

begin
   Put("Centigrade to Farenheight temperature table");
   New_Line(2);
   for Count in INTEGER range -2..12 loop
      X := 10 * Count;
      Y := 32 + X * 9 / 5;
      Put("C =");
      Put(X, 5);
      Put("    F =");
      Put(Y, 5);
      if X = 0 then
         Put("  Freezing point of water");
      end if;
      if X = 100 then
         Put("  Boiling point of water");
      end if;
      New_Line;
   end loop;
end;




-- Result of execution

-- Centigrade to Farenheit temperature table
--
-- C =  -20    F =   -4
-- C =  -10    F =   14
-- C =    0    F =   32  Freezing point of water
-- C =   10    F =   50
-- C =   20    F =   68
-- C =   30    F =   86
-- C =   40    F =  104
-- C =   50    F =  122
-- C =   60    F =  140
-- C =   70    F =  158
-- C =   80    F =  176
-- C =   90    F =  194
-- C =  100    F =  212  Boiling point of water
-- C =  110    F =  230
-- C =  120    F =  248
```

Compile and run e_c05_p6.ada, then examine the program named e_c05_p7.ada for an example that uses some poor choices of variable names and omits the comments. This program is the same program that you just studied because it does exactly the same thing, but I am sure you will find it a little harder to follow since the names are not very useful in helping you understand what it is doing. In the beginning of this tutorial, we studied a program named e_c02_p1.ada which was absolutely ridiculous in style. No serious programmer would write such a program as that, but many programmers seem to be happy using a style similar to the present example.

You should begin, even at this early point in your Ada programming development, to develop good formatting skills. Remember that Ada was designed to be written once and read many times, so the extra time spent keying in long meaningful identifiers will be worth it to your colleagues when they find a need to study your code.

PROGRAMMING EXERCISES

1. Write a program with looping and conditional statements that will output the year and your age during each year from the time you were born until you were 21. Include a special note the year you started school and another note the year you graduated from High School. Example output follows;(Solution)

```
          In 1938, I was 0 years old.
          In 1939, I was 1 years old.
               ...
               ...
          In 1943, I was 5 years old, and started School.
               ...
               etc.
```

```
                                    -- Chapter 5 - Programming exercise 1

with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure Ch05_1 is

   Year : INTEGER;

begin
   for Age in 0..21 loop
      Put("In");
      Put(Age + 1938,5);
      Put(", I was");
      Put(Age,3);
      Put(" years old");
      if Age = 5 then
         Put(", and started school");
      end if;
      if Age = 17 then
         Put(", and graduated from high school");
      end if;
      Put(".");
      New_Line;
   end loop;
end Ch05_1;




-- Result of execution

-- In 1938, I was  0 years old.
-- In 1939, I was  1 years old.
-- In 1940, I was  2 years old.
-- In 1941, I was  3 years old.
-- In 1942, I was  4 years old.
-- In 1943, I was  5 years old, and started school.
-- In 1944, I was  6 years old.
-- In 1945, I was  7 years old.
-- In 1946, I was  8 years old.
-- In 1947, I was  9 years old.
-- In 1948, I was 10 years old.
-- In 1949, I was 11 years old.
-- In 1950, I was 12 years old.
-- In 1951, I was 13 years old.
-- In 1952, I was 14 years old.
-- In 1953, I was 15 years old.
-- In 1954, I was 16 years old.
-- In 1955, I was 17 years old, and graduated from high school.
-- In 1956, I was 18 years old.
-- In 1957, I was 19 years old.
-- In 1958, I was 10 years old.
-- In 1959, I was 21 years old.
```

2. Rewrite exercise 1 using a case statement.(Solution)

```
                              -- Chapter 5 - Programming exercise 2
with Ada.Text_IO, ADa.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure Ch05_2 is

   Year : INTEGER;

begin
   for Age in 0..21 loop
      Put("In");
      Put(Age + 1938,5);
      Put(", I was");
      Put(Age,3);
      Put(" years old");
      case Age is
         when  5 => Put(", and started school");
         when 17 => Put(", and graduated from high school");
         when others => null;
      end case;
      Put(".");
      New_Line;
   end loop;
end Ch05_2;




-- Result of execution

-- In 1938, I was  0 years old.
-- In 1939, I was  1 years old.
-- In 1940, I was  2 years old.
-- In 1941, I was  3 years old.
-- In 1942, I was  4 years old.
-- In 1943, I was  5 years old, and started school.
-- In 1944, I was  6 years old.
-- In 1945, I was  7 years old.
-- In 1946, I was  8 years old.
-- In 1947, I was  9 years old.
-- In 1948, I was 10 years old.
-- In 1949, I was 11 years old.
-- In 1950, I was 12 years old.
-- In 1951, I was 13 years old.
-- In 1952, I was 14 years old.
-- In 1953, I was 15 years old.
-- In 1954, I was 16 years old.
-- In 1955, I was 17 years old, and graduated from high school.
-- In 1956, I was 18 years old.
-- In 1957, I was 19 years old.
-- In 1958, I was 10 years old.
-- In 1959, I was 21 years old.
```

Ada Tutorial - Chapter 6

ADDITIONAL SCALAR TYPES

OUR OWN TYPES

Most of the example programs in this tutorial have used the predefined type **INTEGER** for illustrating various concepts, and it is an excellent choice due to its versatility. There are several other types available because they are part of the Ada definition, and we can define our own types for special purposes. This chapter will illustrate some of the reasons for doing so.

A complete description of types will be given in the next chapter but first we will learn how to use some of them. We are caught in a dilemma like the proverbial, "which came first, the chicken or the egg?", and must select which to define first. We have chosen to illustrate usage in this chapter, and then give the details of type definition in the next chapter.

A FEW INTEGER CLASS TYPES

Example program ------> **e_c06_p1.ada**

```
                                       -- Chapter 6 - Program 1
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure AllInt is

   Data  : INTEGER;
   Form  : POSITIVE;
   Once  : NATURAL;

   type    MY_INTEGER is range -1000..24000;
   type    MY_SHORT   is range -12..127;
   subtype MY_SUBTYPE is MY_INTEGER range -12..127;

   Index : MY_INTEGER := 345;
   Stuff : MY_INTEGER := 33;
   Count : MY_SHORT := 54;

begin
   Put("The type MY_SHORT covers the range of");
   Data := INTEGER(MY_SHORT'FIRST);
   Put(Data);
   Put(" to");
   Data := INTEGER(MY_SHORT'LAST);
   Put(Data);
   New_Line;
   Put("and its base covers the range of");
   Data := INTEGER(MY_SHORT'BASE'FIRST);
   Put(Data);
   Put(" to");
   Data := INTEGER(MY_SHORT'BASE'LAST);
   Put(Data);
   New_Line(2);

   Put("The type MY_INTEGER covers the range of");
   Put(INTEGER(MY_INTEGER'FIRST));
   Put(" to");
   Put(INTEGER(MY_INTEGER'LAST));
   New_Line;
   Put("and its base covers the range of");
   Put(INTEGER(MY_INTEGER'BASE'FIRST));
   Put(" to");
   Put(INTEGER(MY_INTEGER'BASE'LAST));
   New_Line(2);

   if Index in MY_SUBTYPE then
      Put_Line("Index is in the range of MY_SUBTYPE");
   end if;

   if Index not in MY_SUBTYPE then
      Put_Line("Index is not in the range of MY_SUBTYPE");
   end if;

   if Index in 12..377 then
      Put_Line("Index is in the range of 12..377");
   end if;

   if Index not in Stuff..3 * (Stuff - 4) then
      Put_Line("Index is not in the range of Stuff..3 * (Stuff - 4)");
   end if;

end AllInt;




-- Result of execution

-- The type MY_SHORT covers the range of        -12 to        127
-- and its base covers the range of       -128 to        127
--
-- The type MY_INTEGER covers the range of      -1000 to      24000
-- and its base covers the range of     -32768 to      32767
--
-- Index is not in the range of MY_SUBTYPE
-- Index is in the range of 12..377
-- Index is not in the range of Stuff..3 * (Stuff - 4)
```

Examine the file named e_c06_p1.ada for some examples of how and why we might use our own type definitions. The three types illustrated in lines 7 through 9 are available with all Ada compilers because they are so versatile and useful, and because they are required by the Ada programming language. As we have said before, the type **INTEGER** defines a variable that can have any whole value between -2,147,483,648 to 2,147,483,647 on most 32 bit computers. It should be pointed out that some microcomputers use a much smaller range as standard. The type **NATURAL** defines a variable from 0 to 2,147,483,647, and the type **POSITIVE** covers the range from 1 to 2,147,483,647 on most 32 bit computers.

Consider the word "most" in the last paragraph, and think about the problems you could have if you wrote a program that depended on a particular variable covering the listed range, and tried to move the program to a different machine which used a different range. You could be faced with a large rewrite problem in order to get the program to work on the new computer.

HOW CAN WE HELP SOLVE PORTABILITY PROBLEMS

Suppose we defined our type to cover a certain range, such as illustrated in line 11 of this example program, and moved the program to another computer. According to the definition of Ada, the new compiler would be obligated to create a type for us that would cover the given range, or give us a compile time error telling us that the hardware simply could not support the defined range. In this case, due to the rather small range requested, any meaningful compiler and machine combination would be able to cover the defined range, and we would have a program that would run in spite of differences in the way the standard types were defined. Good programming practice, especially if the source code may need to be moved to other computers, would define all ranges explicitly and avoid the implementation defined limits built into the compiler.

Two new types are defined in lines 11 and 12, and the program uses some new attributes to illustrate the new types. In lines 21 and 24, we use two attributes which we have used before, but in lines 28 and 31, we use two new attributes. In order for the system to create a type which covers a range of -1000 to 24000, it must use a structure with enough binary bits to cover the given range. The range is not composed of binary limits so the system will have to define enough bits to cover this range and a little more. It will probably define some number of 8-bit bytes and the range covered by the full pattern, as defined, is called the base range. The two new attributes give the limits of the base selected by the system. The base limits will probably be -32768 to 32767 even if you are using a 32 bit system or it may even use -128 to 127.

DO YOU HAVE A SMART COMPILER?

The type illustrated here named **MY_SHORT** has defined limits of -12 and 127, a relatively small range. It is small enough that it can fit into a base range of -128 to 127, which could be stored in a single 8-bit byte. If your compiler is smart enough to realize that, it could use a single 8-bit byte to store every variable of this type, and if you had a lot of these to store, it would save you a lot of memory. You will probably find however, that most compilers will simply use the full **INTEGER** range for the base type of even this small number. Four attributes of two different types are displayed on the monitor for your information. You can see from the results of running this program exactly how your compiler stores these two types.

THE in AND not in OPERATORS

We have a new operator to learn about now, the **in** operator illustrated in line 46. If the variable **Index**, which has a current value of 345 due to initialization, is within the defined range of the subtype **MY_SUBTYPE**, a **BOOLEAN** type **TRUE** will be returned, otherwise a **BOOLEAN** type **FALSE** is returned. This result can be assigned to a **BOOLEAN** variable or used for a boolean decision as shown. In this case, the value of **Index** is not in the range of **MY_SUBTYPE** so a **FALSE** is returned and the message will not be output. Another operation is illustrated in line 50 which is the **not in** operation, and should be self explanatory. You should be able to see that the message in line 51 will be displayed. The **in** and **not in** operators are further illustrated in lines 54 and 58 where an explicit range is used for the test range.

Be sure to compile and run this program and observe the output. Here is a chance for you to see if you have a smart compiler. If you check the package named **Interfaces**, you will find that you have a few other types available such as **INTEGER_8**, **INTEGER_16**, and **INTEGER_32**. These are all optional, but some will surely be available with your compiler.

THE ENUMERATED TYPE

Example program ------> **e_c06_p2.ada**

```
                                                -- Chapter 6 - Program 2
with Ada.Text_IO;
use Ada.Text_IO;

procedure Enum is

   type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);
   subtype WORK_DAY is DAY range MON..FRI;
   subtype PLAY_DAY is DAY range SAT..SUN;

   type HEAVENLY_BODY is (MOON, SUN, EARTH, MARS);
   Big_Sphere : HEAVENLY_BODY;
   package Day_IO is new Ada.Text_IO.Enumeration_IO(DAY);
   use Day_IO;

   package Body_IO is new Ada.Text_IO.Enumeration_IO(HEAVENLY_BODY);
   use Body_IO;

   Day_Of_Week : DAY;
   Today       : DAY;
   Happy_Day   : PLAY_DAY;
   Bowling_Day : DAY range THU..SAT;
   Index       : INTEGER;

begin

   Day_Of_Week := WED;                        -- WED
   Day_Of_Week := DAY'FIRST;                  -- MON
   Day_Of_Week := DAY'LAST;                   -- SUN
   Day_Of_Week := DAY'PRED(Day_Of_Week);      -- SAT
   Day_Of_Week := DAY'SUCC(PLAY_DAY'FIRST);   -- SUN
   Index := DAY'POS(MON);                     -- 0
   Index := DAY'POS(WED);                     -- 2
   Day_Of_Week := DAY'VAL(1);                 -- TUE

   for Day_Of_Week in WORK_DAY loop
      Put("We are in the workday loop");
      New_Line;
   end loop;

   Today := THU;
   if Today <= WED then
      Put("Early in the week");
      New_Line;
   end if;

   if Today >= WED then
      Put("Late in the week");
      New_Line;
   end if;

   Today := SUN;
   Big_Sphere := SUN;

   Today := DAY'(SUN);
   Big_Sphere := HEAVENLY_BODY'(SUN);

   Put(Today);
   Put(DAY'PRED(Today));
   Put_Line(" from type DAY.");
   Put(Big_Sphere);
   Put(HEAVENLY_BODY'PRED(Big_Sphere));
   Put_Line(" from type HEAVENLY_BODY");

end Enum;




-- Result of execution

-- We are in the workday loop
-- We are in the workday loop
-- We are in the workday loop
-- We are in the workday loop
-- We are in the workday loop
-- Late in the week
-- SUNSAT from type DAY.
-- SUNMOON from type HEAVENLY_BODY
```

Examine the program named e_c06_p2.ada for our first look at an enumerated type and how it is used in a program. Line 7 is the first definition of an enumerated type, and uses the reserved words **type** and **is** as shown. The type name is given between the two reserved words and the values which a variable of this type is allowed to have assigned to it are given as a list within parentheses. The values actually represent numerical values from 0 up to that value required for the largest value, in this case 6, since the numbering will be assigned from 0 to 6. In line 20, the variable named **Day_Of_Week** is declared to be of type **DAY**, so it can be assigned any of the 7 values listed for the type **DAY**, and no others. We could assign the values 0, 1, 2,.. 6 to represent the 7 days of the week and use the numerical values within the program, but by using the enumerated type, we can refer to Sunday as **SUN**, Monday as **MON**, etc., making the program much clearer and easy to follow.

There are three predefined enumerated types in the Ada language, **BOOLEAN**, **CHARACTER**, and **WIDE_CHARACTER**. The **BOOLEAN** type is an enumerated type with two possible values, but it has some special properties available with no other enumerated variables. These will be discussed using the next example program.

Jumping ahead to the executable code in the current example program, we illustrate assignment in line 28, where we assign the value of **WED** to the variable **Day_Of_Week**. Lines 29 and 30 illustrate the **FIRST** and **LAST** attributes which we have seen before for **INTEGER** type variables. Just as -2,147,483,648 is the lowest possible value that can be assigned to a 32 bit **INTEGER** type variable, **MON** is the lowest, and hence the first, value that can be assigned to a variable of type **DAY**.

TWO NEW ATTRIBUTES

Lines 31 and 32 illustrate the attributes **PRED**, which means the predecessor, and **SUCC**, which means the successor. **PRED** returns the value of the predecessor of the present value of the variable used as an argument. Since the variable **Day_Of_Week** was assigned the value of **SUN** in line 30, and the day just prior to **SUN** is **SAT**, **SAT** is assigned to the variable **Day_Of_Week** in line 31. It is an error to attempt to take the **PRED** of a variable which contains the first value in the available list, and will result in raising the exception **Range_Error**. Likewise, an attempt to take the **SUCC** of any variable that is at its maximum value will result in the exception **Range_Error** being raised. Exceptions will be covered in detail later in this tutorial. At this time simply remember that an exception refers to an exceptional condition or an error.

WHAT IS A SUBTYPE OF AN ENUMERATED TYPE?

In lines 8 and 9, we define two subtypes of the type **DAY** which will have all the characteristics of type **DAY** except for a more restricted range. A variable that is declared to be of type **PLAY_DAY** can be assigned either of two values, **SAT** or **SUN**. **SAT** will have a numerical value of 5, and **SUN** will have a numerical value of 6, both of these being inherited from the parent type, **DAY**. Thus in line 32, we use the attribute **FIRST** to get the first day of type **PLAY_DAY**, which will be **SAT**, then use the attribute **SUCC** to get the successor of that value, which will be **SUN**. Notice how the attributes can be combined to obtain the needed information. A subtype is assignment compatible with its parent type. We will discuss subtypes in greater detail in the next chapter of this tutorial.

NOW FOR THE POS AND VAL ATTRIBUTES

The **POS** attribute will return a value of type universal_integer, the value representing the position of the enumerated value within the parentheses as shown in lines 33 and 34. The **VAL** attribute will return the enumerated value of the numerical value included in the parentheses. Notice that if the type **DAY** in line 35 were changed to **PLAY_DAY**, an error would be returned, since that is an illegal enumerated value for that type. The error would be returned by raising the exception **Range_Error**.

WHAT ABOUT ENUMERATED ASSIGNMENTS?

The value of **Happy_Day** can be assigned to **Day_Of_Week** at any time because they are type compatible, and any value that can be legally assigned to **Happy_Day** can also be assigned to **Day_Of_Week**. **Day_Of_Week** cannot always be assigned to **Happy_Day** however, because **Day_Of_Week** is permitted to contain some values which are not legal to assign to **Happy_Day**. This would illustrate that some care must be exercised when using the enumerated type, but if used properly, it can help in program debugging by the use of the strong type-checking defined into the Ada language.

USING ENUMERATED TYPES FOR CONTROL

The loop in lines 37 through 40 covers exactly the range covered by the subtype **WORK_DAY**, so we can use it in the range part of the definition of the loop. When you run this program, you will see that the loop will be executed exactly five times.

Lines 42 through 51 contain two relational checks on the variable **Today** to illustrate that the enumerated type variable can be used in a **BOOLEAN** expression. All of the boolean operators are available, which includes the following list, and no others;

```
     =     equality
     /=    inequality
     >     greater than
     >=    greater than or equal to
     <     less than
     <=    less than or equal to
```

No mathematical operations are available with enumerated type variables. Assignments are available as illustrated in the present example program.

WHAT IS QUALIFICATION?

In lines 53 and 54, we assign the same value to two different enumerated type variables. At least it seems to be the same value. In actuality, they are two different values with the same name, namely **SUN**. Because Ada does such strong type checking, it is smart enough to realize that they are actually two different constants and it will select the one that it needs for each statement based on the type of the variable to which it will be assigned.

Lines 56 and 57 make the identical assignments by qualifying which value you are interested in, but in this case, the qualifications are unnecessary. There could be a case when you would need to tell the system which value of **SUN** you are interested in. Qualification uses the type followed by a "tick", or apostrophe, prior to the enumeration value.

OUTPUTTING ENUMERATED VARIABLES

The statements in lines 59 and 60 output the current value of the variable **Today**, and the predecessor of the current value. Finally, the same values are output for the variable **Big_Sphere**, and when you run the program, you will see that the same value is output for the first value in each line, but the second values differ for the two variables. Note the four extra lines of code given in program lines 14 through 18. These are used to tell the system how to output enumerated variables, and we will cover the essentials of how this works very soon.

The **in** and **not in** operators which we studied in the last program are available for use with the enumerated type variable. In fact, they are available with all discrete types. Be sure to compile and run this program and after studying the results, see if you can modify the program to output additional enumerated values.

THE BOOLEAN VARIABLE

Example program ------> **e_c06_p3.ada**

```
                                        -- Chapter 6 - Program 3
with Ada.Text_IO;
use Ada.Text_IO;

procedure BoolVars is

   package Bool_IO is new Ada.Text_IO.Enumeration_IO(BOOLEAN);
   use Bool_IO;

   Correct  : BOOLEAN;
   Maybe    : BOOLEAN;
   Probably : BOOLEAN;

begin

   Correct := TRUE;                         -- TRUE
   Maybe := FALSE;                          -- FALSE
   Probably := Correct or Maybe;            -- TRUE
   Probably := Correct and Maybe;           -- FALSE
   Probably := Correct xor Maybe;           -- TRUE
   Probably := Correct and not Maybe;       -- TRUE
   Probably := BOOLEAN'FIRST;               -- FALSE
   Probably := BOOLEAN'LAST;                -- TRUE
   if Maybe < Correct then
      Put("FALSE is of less value than TRUE in a BOOLEAN variable");
      New_Line;
   end if;

   Put(Correct, 8);
   Put(Maybe, 8);
   New_Line;

end BoolVars;




-- Result of execution

-- FALSE is of less value than TRUE in a BOOLEAN variable
-- TRUE    FALSE
```

The program named e_c06_p3.ada is an illustration of how to use the **BOOLEAN** variable, which is actually a special case of an enumerated variable. It is simply an enumerated type with two possible values, **TRUE** or **FALSE**. Since it is an enumerated type, all of the operations available with the enumerated type are available with the **BOOLEAN** type, including all six of the relational operators, the assignment operator, the attributes, and no mathematical operators.

THE LOGICAL OPERATORS

The **BOOLEAN** type has some of its own unique operators that are available with no other types, and these are the logical operators. The logical operators were defined earlier, but are repeated here as a complete list.

```
     and        logical and operation
     or         logical or operation
     xor        exclusive or of two values
     not        inversion of the value
     and then   short circuit and operation
     or else    short circuit or operation
```

It should be pointed out that **FALSE** is of less numerical value than **TRUE**, by definition, and in actuality, the value of **FALSE** is 0, and the value of **TRUE** is 1.

The program illustrates how to output **BOOLEAN** values in lines 29 and 30, after the package instantiation in lines 7 and 8. Notice that the **Enumeration_IO** library is used for **BOOLEAN** output illustrating again that **BOOLEAN** is a special case of the enumerated type.

Be sure to compile and execute this program to see that it really does compile and execute as stated. Someday, you will need the ability to display the **BOOLEAN** results as is done in this program.

SOME USELESS ATTRIBUTES OF INTEGER TYPES

Example program ------> **e_c06_p4.ada**

```
                                        -- Chapter 6 - Program 4
with Ada.Text_IO;
use Ada.Text_IO;

procedure IncrInt is

   Index : INTEGER;

begin

   Index := 13;
   Index := INTEGER'POS(Index);          -- Still 13
   Index := INTEGER'VAL(Index);          -- Still 13
   Index := INTEGER'SUCC(Index);         -- Incremented to 14
   Index := INTEGER'PRED(Index);         -- Decremented to 13

   Index := Index + 1;       -- preferred method of incrementing

end IncrInt;




-- Result of execution

--   (No output data from this program.)
```

It may seem silly to illustrate some useless attributes but the program named e_c06_p4.ada does that very thing. The **POS** attribute of an integer variable is defined as being the number itself, and the **VAL** is also the number. The **SUCC** of an integer variable is the next number, and the **PRED** is the predecessor. These last two attributes could be useful for incrementing or decrementing a variable in a program, but good programming practice would forbid such a use of these attributes. You should use the very clear and easy to understand method of adding one to the value and assigning the result back to the variable, as illustrated in line 17 of the program.

Even though these are really useless at this point in time, the fact that this can be done will be very useful when we get to the study of generics later in this tutorial.

Compile and run this program, adding some output to gain some of your own programming experience.

MODULAR TYPE VARIABLES

Example program ------> **e_c06_p5.ada**

```
                                                -- Chapter 6 - Program 5
with Ada.Text_IO;
use Ada.Text_IO;

procedure Modular is

   type DIAL_RANGE is mod 5;
   Dial : DIAL_RANGE := 3;

   type MY_BINARY_BIT is mod 2;
   My_Bit : MY_BINARY_BIT := 1;

   type MY_UNSIGNED_SHORT_INT is mod 65536;
   type MY_UNSIGNED_BYTE      is mod 256;

   package Mod_IO is new Ada.Text_IO.Modular_IO(DIAL_RANGE);
   use Mod_IO;

   package Bit_IO is new Ada.Text_IO.Modular_IO(MY_BINARY_BIT);
   use Bit_IO;

begin

   for Index in 1..6 loop
      Dial := Dial + 1;
      Put("The value of Dial is ");
      Put(Dial);
      Put(", and the binary bit is ");
      Put(My_Bit);
      My_Bit := My_Bit + 1;
--    My_Bit := My_Bit + 2;   -- Error, 2 is too big to add to My_Bit
      New_Line;
   end loop;

   New_Line;

   for Index in 1..6 loop
      Dial := Dial - 1;
      Put("The value of Dial is ");
      Put(Dial);
      Put(", and the binary bit is ");
      Put(My_Bit);
      My_Bit := My_Bit - 1;
      New_Line;
   end loop;

end Modular;




-- Result of execution
--
--- The value of Dial is 4, and the binary bit is 1
--- The value of Dial is 0, and the binary bit is 0
--- The value of Dial is 1, and the binary bit is 1
--- The value of Dial is 2, and the binary bit is 0
--- The value of Dial is 3, and the binary bit is 1
--- The value of Dial is 4, and the binary bit is 0

--- The value of Dial is 3, and the binary bit is 1
--- The value of Dial is 2, and the binary bit is 0
--- The value of Dial is 1, and the binary bit is 1
--- The value of Dial is 0, and the binary bit is 0
--- The value of Dial is 4, and the binary bit is 1
--- The value of Dial is 3, and the binary bit is 0
```

Ada 95 introduced a new type into the language, the modular type variable which can be very useful in certain kinds of programs. In line 7, we define the type **DIAL_RANGE** as **mod** 5, which means that any variable declared to be of this type will "modulo" when it reaches the value of 5. The variable declared in line 8, namely **Dial**, can store only values in the range of 0 to 4, and when it is incremented past 4 it modulo's back to zero with no error indication of overflow. Any value from 0 to 4 can be added to it, or subtracted from it and it will modulo at either the upper or lower end of it's permitted range. A modulo variable cannot store a negative number. It is essentially an unsigned **INTEGER** with the added provision that it cannot overflow or underflow.

The type declared in line 10 is another **mod** variable named **MY_BINARY_BIT** which is only permitted to store a value of 0 or 1, and a variable of this type named **My_Bit** is declared in line 11.

Two other types are defined in lines 13 and 14 for illustration, but they are not actually used in this program.

In lines 16 through 20, we instantiate modulo output packages for these variable types and use them in the main program. We use an incrementing loop, followed by a decrementing loop to illustrate both overflow and underflow with these two new types. The program should be very easy for you to follow and understand, so nothing more needs to be said about it.

There are some predefined modular types available such as **UNSIGNED_8**, **UNSIGNED_16**, and **UNSIGNED_32**. They are defined in the package **Interfaces** along with several bitwise **and**, **or**, **xor**, etc. operations, and several shifting and rotating subprograms.

FLOATING POINT VARIABLES

Example program ------> **e_c06_p7.ada**

```
                                        -- Chapter 6 - Program 7
with Ada.Text_IO;
use Ada.Text_IO;

procedure FloatVar is

   PI     : constant := 3.1416;
   TWO_PI : constant := 2 * PI;

   R : FLOAT;

   type MY_FLOAT is digits 7;
   type MY_LONG_FLOAT is digits 15;

   Area   : MY_FLOAT := 2.345_12e4;      -- This is 23451.2
   Length : MY_FLOAT := 25.123;
   Factor : MY_FLOAT := 8.89;
   Cover  : MY_LONG_FLOAT;
   What   : BOOLEAN;
   Index  : INTEGER := 4;

   package Int_IO is new Ada.Text_IO.Integer_IO(INTEGER);
   use Int_IO;
   package Flt_IO is new Ada.Text_IO.Float_IO(MY_FLOAT);
   use Flt_IO;
```

```
begin
                           -- Arithmetic float operations
   Area := Length + Factor + 12.56;
   Area := Length - Factor - 12.56;
   Area := Length * Factor * 2#111.0#;   -- this is decimal 7.0
   Area := Length / Factor;
   Area := Length ** 3;
   Area := Length ** (-3);
   Area := Length ** Index;

                           -- Arithmetic logical compares
   What := Length =  Factor;
   What := Length /= Factor;
   What := Length >  Factor;
   What := Length >= Factor;
   What := Length <  Factor;
   What := Length <= Factor;

   Area := 0.0031 + (0.027_3 * TWO_PI) / (Length ** (-Index/2));
   Cover := 27.3 * TWO_PI * MY_LONG_FLOAT(Area);

   Put("Area is now ");
   Put(Area);
   Put(Area,5);
   New_Line;

   Put("Area is now ");
   Put(Area,5,5);
   Put(Area,5,5,0);
   New_Line;

   Put("MY_FLOAT'DIGITS =      ");
   Put(MY_FLOAT'DIGITS);
   New_Line;
   Put("MY_FLOAT'BASE'FIRST =  ");
   Put(MY_FLOAT'BASE'FIRST);
   New_Line;
   Put("MY_FLOAT'BASE'LAST =   ");
   Put(MY_FLOAT'BASE'LAST);
   New_Line;

end FloatVar;




-- Result of execution

-- Area is now  1.082677E+02    1.082677E+02
-- Area is now     1.08268E+02  108.26771
-- MY_FLOAT'DIGITS =                7
-- MY_FLOAT'BASE'FIRST =  -1.797693E+308
-- MY_FLOAT'BASE'LAST =    1.797693E+308
```

Examine the program named e_c06_p7.ada for examples of nearly all operations possible with floating point numbers.

We begin, in lines 7 and 8, by defining two constants, the second being defined in terms of the first. Remember that any thing used in an Ada program must be previously defined, and you will know all of the rules for defining a value in terms of some other value. The two constants are of type universal_real, so they can be used with any of the various real types we will encounter in this program. We declare a variable named **R** in line 10 of type **FLOAT**, which is defined by the compiler writer, then two new types of floating point numbers, and we finally declare six variables of various types.

Two additional floating point types, **SHORT_FLOAT** and **LONG_FLOAT** are defined as optional by the ARM and may be available with your compiler. You can find out by checking the documentation supplied with your compiler or by declaring variables of those types to see if the compiler will accept the declarations. If they do exist, you can determine their limits by using attributes as defined below. You can also determine their limits by studying Annex M which is required to be supplied with every Ada 95 compiler. Annex M contains the definition of all implementation dependent entities.

HOW TO DECLARE A NEW FLOATING POINT TYPE

The reserved word **digits** used in line 12 tells the compiler that you don't care how many bytes of storage it uses to define the number, but it must store at least 7 significant digits for every variable of type **MY_FLOAT**. Line 13 requests a minimum of 15 significant digits for every variable of type **MY_LONG_FLOAT**. The type **FLOAT** as used in line 10, on the other hand, only requires that it be a floating point number, and the compiler writer has the option of using as many significant digits as he desires to implement variables of this type. If you wrote a program that ran well with one compiler, it may not run properly with a different compiler, either because the new one did not use enough significant digits, or because the new one used far more, causing your program to run out of storage space or run too slowly. The forms in lines 12 and 13 are therefore preferred for portability purposes. More will be said about declaring floating point types in the next chapter of this tutorial.

FLOATING POINT LITERALS

The distinguishing characteristic that defines a floating point number is the use of a decimal point. Ada requires at least one digit before and after the decimal point in all floating point literals, although either or both may be a zero. Single embedded underlines are allowed to improve readability, but cannot be adjacent to the decimal point. The underlines are ignored by the compiler, and have no significance. Any radix from 2 through 16 may be used by first giving the radix, then enclosing the number in pound (#) characters. The base 10 is the default and need not be specified. Exponential notation can be used, the exponent being to the same base as that indicated by the radix. A binary floating point literal is illustrated in line 31 of the program and you can see that the radix is similar to that used for integer class literals.

FLOATING POINT MATHEMATICAL OPERATORS

Lines 29 through 35 illustrate the mathematical operators available with floating point variables and should be self explanatory with the exception of the exponential operator. This can use only an integer type of exponent but it can be either positive or negative. Of course, zero is also permissible.

All six logical comparisons are available with floating point variables as illustrated in lines 38 through 43. Keep in mind that it is bad practice to compare two floating point numbers for equality or inequality in any language, but it can be done in Ada. The next two lines, 45 and 46, illustrate some multiple mathematical operations.

As with all variables, the types must agree within all mathematical and logical operations and the result must be assigned to the right type of variable, or a type error will be generated at compile time. In line 46, the variable **Area** must be transformed in type prior to being assigned to **Cover** since they are of different types. The entire statement will be evaluated as of type **MY_LONG_FLOAT**, since that will be the final result. The constant 27.3, and the constant **TWO_PI**, will be transformed automatically from universal_real to **MY_LONG_FLOAT** prior to the multiplications.

NOW TO OUTPUT SOME FLOATING POINT VALUES

In lines 22 through 25 we instantiate a copy of the **Integer_IO** package and a copy of the **Float_IO** package for use with the types **INTEGER** and **MY_FLOAT** and use it in lines 49 through 55. The variable **Area** will be output in a default exponential notation in line 49, but with 5 digits prior to the decimal point in line 50. Line 54 adds an additional 5, which will cause 5 digits to be output following the decimal point, and the fourth field, in this case a zero, causes the output to be written with a zero exponent, or no exponential notation.

FLOATING POINT ATTRIBUTES

Floating point variables and types are no different from the scalar types concerning attributes available for your use, except that there are different attributes available. Lines 58 through 66 illustrate the use of some of the available attributes. The attribute named **DIGITS**, gives the number of significant digits available with the specific type, and the return is a universal_integer type.

The attributes named **SMALL** and **LARGE** give the smallest and largest numbers available with the corresponding type, and the attributes named **FIRST** and **LAST** combined with the **BASE** attribute as shown in lines 62 and 65, define the extreme values as used by the underlying base type of the actual user's type. All four of these attributes return a value of the type universal_real, and are displayed on the monitor for your information. Note that there are other attributes available with the floating point type, but only these will be elaborated upon at this time.

See Annex K of the ARM for additional information on attributes. This appendix lists all of the attributes available with an Ada system.

Compile and run this program and observe the output. The actual output is the clearest description of the **Put** procedure when used with floating point numbers. Study the result of the attribute outputs before continuing on to the next example program.

FIXED POINT VARIABLES

Example program ------> e_c06_p8.ada

```
                                       -- Chapter 6 - Program 8
with Ada.Text_IO;
use Ada.Text_IO;

procedure Fixed is

   COARSE_PI : constant := 3.15;

   type MY_FIXED is delta 0.1  range -40.0..120.0;
   type ANGLE    is delta 0.05 range -COARSE_PI..COARSE_PI;

   Theta, Omega, Phi : ANGLE := 0.50;
   Funny_Value : MY_FIXED;

   package Int_IO is new Ada.Text_IO.Integer_IO(INTEGER);
   use Int_IO;
   package Fix_IO is new Ada.Text_IO.Fixed_IO(MY_FIXED);
   use Fix_IO;
   package Fix2_IO is new Ada.Text_IO.Fixed_IO(ANGLE);
   use Fix2_IO;
   package Flt_IO is new Ada.Text_IO.Float_IO(FLOAT);
   use Flt_IO;

begin

   Put("Theta starts off with the value");
   Put(Theta, 5, 2, 0);
   New_Line(2);
   Theta := Omega + Phi;
   Theta := Omega - Phi;
   Theta := 5 * Omega - 2 * Phi;
   Theta := ANGLE(Omega * Phi);
   Theta := ANGLE(3 * Omega / Phi);
   Theta := abs(Omega - 3 * Phi);

   Funny_Value := 5.1;
   for Index in 1..10 loop
      Put("Funny_Value is now");
      Put(Funny_Value, 5, 1, 0);
      Put(Funny_Value, 5, 5, 0);
      Funny_Value := MY_FIXED(Funny_Value * MY_FIXED(1.1));
      New_Line;
   end loop;

   New_Line;
   Put("MY_FIXED'DELTA = ");
   Put(MY_FIXED(MY_FIXED'DELTA));
   New_Line;
   Put("MY_FIXED'FIRST = ");
   Put(MY_FIXED'FIRST);
   New_Line;
   Put("MY_FIXED'LAST =  ");
   Put(MY_FIXED'LAST);
   New_Line;

end Fixed;




-- Result of execution

-- Theta starts off with the value    0.50
--
-- Funny_Value is now    5.1    5.10156
-- Funny_Value is now    5.6    5.61719
-- Funny_Value is now    6.2    6.18750
-- Funny_Value is now    6.8    6.81250
-- Funny_Value is now    7.5    7.50391
-- Funny_Value is now    8.3    8.26562
-- Funny_Value is now    9.1    9.10156
-- Funny_Value is now   10.0   10.02344
-- Funny_Value is now   11.0   11.03906
-- Funny_Value is now   12.2   12.16016
--
-- MY_FIXED'DELTA =    0.1
-- MY_FIXED'FIRST =  -40.0
-- MY_FIXED'LAST  =  120.0
```

Fixed point variables are a relatively new concept and may be a bit confusing, but the file named e_c06_p8.ada will illustrate the use of a few fixed point variables. Line 9 defines a fixed point type as having a **range** of -40.0 to 120.0, and a **delta** of 0.1 which means that a variable of this number can only have a value that is accurate to one digit after the decimal point. There are therefore a fixed number of digits before and after the decimal point, hence the name of this type of variable.

A fixed point number will always be exact since it is defined that way. There can never be a gradual accumulation of error with a fixed point variable. In order to completely understand the fixed point type, one would require a complete understanding of numerical analysis, which is beyond the scope of this tutorial. The program before you will illustrate how to use this type, but no attempt will be made to explain why it should be used.

There are no predefined fixed point types, so it is up to the programmer to define every fixed point type needed, as illustrated in lines 9 and 10. The reserved word **delta** denotes a fixed point type and a **range** is required for every fixed point type. Lines 12 and 13 are used to declare a few variables for use in the program, then lines 17 through 20 instantiate the package **Fixed_IO** for use with our two fixed point types.

HOW DO WE USE FIXED POINT TYPES?

Output of a fixed point type uses the same format as that defined for floating point data as shown in line 27. When we come to arithmetical operations, we find some funny rules which we will simply state, and make no attempt to justify. Variables of the same fixed point types can be added and subtracted freely, provided the results are within the defined range, just like floating point type variables. Multiplication by a constant of type universal_integer is permitted, resulting in the same fixed point type we started with. Multiplication of two fixed point variables results in an anonymous type which must be explicitly converted to some predefined type, as illustrated in lines 33 and 34. The only operator available with the fixed types is the **abs** operator.

Many attributes are available with the fixed point type, some of which are illustrated in lines 46 through 55. The attributes named **DELTA**, **SMALL**, and **LARGE**, each return a value which is of type universal_real, and must be converted to the users fixed type, by a type conversion, before the result can be used in the program. Line 48 illustrates the conversion within the **Put** procedure call.

Be sure to compile and run this program and observe the output to see if it conforms to what you think it should do based on the previous discussion. Note that your compiler may not generate identical output as that listed in the result of execution due to different compiler defaults.

ANOTHER NEW TYPE

Example program ------> **e_c06_p9.ada**

```
                                             -- Chapter 6 - Program 9
with Ada.Text_IO;
use Ada.Text_IO;

procedure Decimal is

   type DOLLAR is delta 0.01 digits 8 range 0.00..1_000.00;
   type DIMES  is delta 0.1  digits 6;
   Amount : DOLLAR := 3.00;
   Coins  : DIMES  := 1.20;

   package Dec_IO is new Ada.Text_IO.Decimal_IO(DOLLAR);
   use Dec_IO;
   package Dime_IO is new Ada.Text_IO.Decimal_IO(DIMES);
   use Dime_IO;

begin

   for Index in 1..8 loop
      Amount := Amount + 1.23;
      Put(Amount);
      Coins := Coins + 3.70;
      Put(Coins);
      New_Line;
   end loop;

end Decimal;
```

```
-- Result of execution
--
--     4.23   4.9
--     5.46   8.6
--     6.69  12.3
--     7.92  16.0
--     9.15  19.7
--    10.38  23.4
--    11.61  27.1
--    12.84  30.8
```

A new type was added to the language with the introduction of Ada 96, the decimal type. It looks and acts very much like any other fixed type with the added limitation that it can only use a **delta** that is a power of 10. Every thing that can be done with a fixed type can be done with the decimal type, and all of the limitations of fixed types are applied to the decimal type also.

The decimal type is intended for financial software and has one advantage over the fixed type, and that is the predefined package named **Ada.Text_IO.Editing** which is a part of the optional Information Systems Annex. This package provides formatting monetary values with the dollar sign, commas, and proper placement of the decimal point. The example program illustrates use of the decimal type, but it does not illustrate the use of the optional annex since it is expected to be supported by only a few Ada compilers, those targeted to the banking and financial industry.

MIXING VARIOUS TYPES

Example program ------> **e_c06_p10.ada**

```
                                        -- Chapter 6 - Program 10
with Ada.Text_IO;
use Ada.Text_IO;

procedure MixTypes is

   PI : constant := 3.1416;
   TWO_PI : constant := 2 * PI;

   type MY_FLOAT is digits 7;
   Size : MY_FLOAT;

   type MY_FIXED is delta 0.1 range -40.0..120.0;
   Temperature : MY_FIXED;

   Index : INTEGER;

begin
   Size := TWO_PI;
   Temperature := 2 * TWO_PI;
   Temperature := 3 * Temperature;
   Temperature := MY_FIXED(12.0 * TWO_PI);

   Size := MY_FLOAT(Temperature) + PI;
   Size := MY_FLOAT(Temperature + PI);

   Index := INTEGER(Size + MY_FLOAT(Temperature) + PI);
   Index := INTEGER(Size) + INTEGER(Temperature) + INTEGER(PI);
   Index := INTEGER(Size + PI) + INTEGER(Temperature);

end MixTypes;
```

```
-- Result of execution

--   (No output from this program.)
```

Examine the program named e_c06_p10.ada for examples of using various types together. It is meant to be an illustration of how to combine some of the various types available in Ada. Many type transformations are illustrated by the explicit conversions in this program and should be easy for you to understand. Note especially, that the final result of lines 27, 28, and 29 will not necessarily be the same due to the rounding that takes place at different points in the calculations.

Note that in Ada, conversion from real to integer always rounds rather than truncates. A value midway between the two integer values will always move away from zero. This is carefully specified in Ada 95, but was not defined for Ada 83.

Compile and execute this program to assure yourself that it will compile correctly.

This would be a good time to get a little more familiar with the ARM by examining sections 3.5.4 and 3.5.7 which define the requirements for **INTEGER** and **FLOAT** type variables. You will not understand it all, but you will understand enough of it to make it a profitable task.

PROGRAMMING EXERCISES

1. Write a program to determine if **LONG_INTEGER** and **SHORT_INTEGER** types are available with your compiler. If they are available, use attributes to determine their characteristics.(Solution)

-- Chapter 6 - Programming exercise 1

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

procedure Ch06_1 is

package Long_IO is new Ada.Text_IO.Integer_IO(LONG_INTEGER);
use Long_IO;

package Short_IO is new Ada.Text_IO.Integer_IO(SHORT_INTEGER);
use Short_IO;

Long_Variable  : LONG_INTEGER;
Short_Variable : SHORT_INTEGER;

begin
   Put_Line("For LONG_INTEGER,");
   Put(LONG_INTEGER'FIRST);
   Put_Line(" is the minimum value");
   Put(LONG_INTEGER'LAST);
   Put_Line(" is the maximum value");

   Put_Line("For SHORT_INTEGER,");
   Put(SHORT_INTEGER'FIRST);
   Put_Line(" is the minimum value");
   Put(SHORT_INTEGER'LAST);
   Put_Line(" is the maximum value");
end Ch06_1;




-- Result of execution
-- For LONG_INTEGER
-- -2147483648 is the minimum value
--  2147483647 is the maximum value
-- For SHORT_INTEGER
-- -32768 is the minimum value
--  32767 is the maximum value
```

2. Do the same thing as exercise 1 for the **LONG_FLOAT** and **SHORT_FLOAT** types. (Solution)

-- Chapter 6 - Programming exercise 2

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

procedure Ch06_2 is

package Long_IO is new Ada.Text_IO.Float_IO(LONG_FLOAT);
use Long_IO;

package Short_IO is new Ada.Text_IO.Float_IO(SHORT_FLOAT);
use Short_IO;

Long_Variable  : LONG_FLOAT;
Short_Variable : SHORT_FLOAT;

begin
   Put_Line("For LONG_FLOAT,");
   Put(LONG_FLOAT'FIRST);
   Put_Line(" is the minimum value");
   Put(LONG_FLOAT'LAST);
   Put_Line(" is the maximum value");

   Put_Line("For SHORT_FLOAT,");
   Put(SHORT_FLOAT'FIRST);
   Put_Line(" is the minimum value");
   Put(SHORT_FLOAT'LAST);
   Put_Line(" is the maximum value");
end Ch06_2;




-- Result of execution

-- For LONG_FLOAT
-- ? is the minimum value
-- ? is the maximum value
-- For SHORT_FLOAT
-- ? is the minimum value
-- ? is the maximum value
```

3. Try to take the **PRED** of the first element of an enumerated variable to see what kind of a run-time error you get. Your compiler may be smart enough to warn you if you try to take it

directly (i.e. - by using the first value in the parentheses), so you may need to assign a variable to the first value and take the **PRED** of the variable.(Solution)

```
                                -- Chapter 6 - Programming exercise 3
with Ada.Text_IO;
use Ada.Text_IO;

procedure Ch06_3 is

type MY_ENUM is (YES, NO, MAYBE, WHY, POSSIBLY);

Answer : MY_ENUM := YES;

begin
   Answer := MY_ENUM'PRED(Answer);
end Ch06_3;




-- Result of execution

-- Exception never handled: constraint_error
-- Value 0 out of range 1..4.

-- (This means that since the range of the enumerated variable
--  only covers four values, trying to assign the value of zero,
--  which is one less than the allowed minimum causes us to go
--  out of the allowable range. )
```

DERIVED TYPES

LET'S TALK ABOUT TYPES

We have been talking about types throughout this tutorial, but we've been sort of working our way around them rather than carefully defining the various kinds of types available in Ada. This chapter will be devoted to a full discussion of types. It was not possible to discuss types in full until we covered a bit of material about the various scalar types and how they are used, but with that material behind us, we will give a full treatment to the topic of types.

There are four different ways to declare a type. You can probably guess that we will discuss each of the ways in detail in this chapter. They are listed as follows;

1. Predefined types. These are provided for us by the compiler writer as defined in the Ada 95 Reference Manual (ARM). We have already used several of these.
2. User defined types. We have already defined a few new types in some of the earlier programs.
3. Derived types. These get their name because they are defined in part based on a previously defined type and derive some of their characteristics from those types. We have not yet encountered the derived type.
4. Subtypes. These are usually a subset of another type upon which they are based. We encountered subtypes in the last chapter.

A type defines a set of values and a set of primitive operations. The primitive operations of a type are;

- The intrinsic predefined operations such as assignment, addition, subtraction, etc.
- Any operations inherited from a parent in the case of derived types. We will study derived types in this chapter.
- Subprograms with one or more parameters or a result of the type that are declared within the same package as the type.

PREDEFINED TYPES IN ADA

The ARM requires every Ada compiler to provide several predefined types, including **INTEGER**, **NATURAL**, **POSITIVE**, **FLOAT**, **CHARACTER**, and **STRING**. Several other types are optional and include **LONG_INTEGER**, **SHORT_INTEGER**, **LONG_FLOAT**, and **SHORT_FLOAT**. It will be left to you to study the documentation that came with your compiler and see which of the optional types are included with your compiler package. You will find this information in Annex M. The actual definition of these types is given in the package named **Standard**.

The predefined types can be used as the basis of the declaration of new types for your specific application and is the topic of this chapter.

USER DEFINED TYPES

Example program ------> **e_c07_p1.ada**

```
                                                 -- Chapter 7 - Program 1
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure DerTypes is

   type LITTLE_INT is range -24..17;
   type TINY_INT   is range -3..2;
   type POS_INT    is range 25..38;
   type TINY_POS   is new POS_INT range 25..30;
   type SALAD_INT  is new INTEGER;
   type ANIMAL_INT is new INTEGER;
   type TREE_INT   is new INTEGER range -557..1098;

   Salad    : SALAD_INT;
   Lettuce  : SALAD_INT := 22;
   Tomatoes : SALAD_INT := 14;
   Animals  : ANIMAL_INT;
   Dogs     : ANIMAL_INT := 3;
   Cats     : ANIMAL_INT := 4;
   Trees    : TREE_INT;
   Oak      : TREE_INT := 12;
   Coconut  : TREE_INT := 8;
   Count    : INTEGER;

begin

   Salad := Lettuce + Tomatoes;
   Animals := Dogs + Cats;
   Trees := Oak + Coconut + TREE_INT(Animals);
   Count := INTEGER(Trees) + INTEGER(Salad);

   Salad := SALAD_INT(Dogs) * Tomatoes +
            SALAD_INT(Cats) * SALAD_INT(Oak) +
            SALAD_INT(Count);
   Put("The 1st Salad calculation is ");
   Put(INTEGER(Salad));
   New_Line;

   Salad := SALAD_INT(Dogs * ANIMAL_INT(Tomatoes) +
                      Cats * ANIMAL_INT(Oak) +
                             ANIMAL_INT(Count));
   Put("The 2nd Salad calculation is ");
   Put(INTEGER(Salad));
   New_Line;

end DerTypes;




-- Result of execution

-- The 1st Salad calculation is         153
-- The 2nd Salad calculation is         153
```

Examine the program named e_c07_p1.ada for several examples of user defined types. Lines 7 through 9 each declare an entirely new type. Considering only the predefined type **INTEGER** and these three, we have four types to use in the program, and each type has nothing to do with the other three. Variables of these types cannot be intermixed in any way without the proper type conversion explicitly stated by the programmer. Moreover, variables declared with one of these types cannot be assigned a value that is outside of its declared range. The structure for declaring a user defined integer class type is given as,

```
type <type-name> is range <lower-limit>..<upper-limit>;
```

As stated earlier, the word **range** is a reserved word, and its presence indicates to the compiler that you wish to have a type of the integer class.

Use of the three new types is not illustrated in this example program, but the diligent student will have no trouble using these new types to declare a few variables, then use them in some mathematical statements.

DERIVED TYPES ARE RELATIVELY NEW

Derived types are an entirely new subject, since they are not available in any of the more popular languages that you may have been programming in. Derived types get their name because they are derived from an existing type rather than being an entirely new creation. A derived type has all of the operations available that are available with the parent type but it may have a more limited range than the range of the parent type. The example program has an illustration of this in line 10 where the type **TINY_POS** is derived from the user defined type **POS_INT** but with a slightly tighter allowable range. Any operation that is legal to be performed on a variable of type **POS_INT** is legal for a variable of type **TINY_POS**.

The key to a derived type is the use of the reserved word **new** along with the type from which the new type will be derived. The structure for declaring a user defined derived type is given as,

```
type <type-name> is new <existing-type>;
```

The derived type will be of the same class as the type from which it is derived.

We continue declaring new types in lines 11 through 13 and because these three are entirely new types, each derived from the parent type **INTEGER**, they cannot be added together, subtracted, compared or even assigned to each other without a bit of extra trouble. Each of the seven new types in lines 7 through 13 share all of the operations that are defined for **INTEGER**, including the arithmetic, logical, and assignment operations, but the operations can only be performed as long as the types of the objects are consistent. Of course, the explicit type conversion can be done to allow the combination of variables of any of these types.

The type **TREE_INT**, defined in line 13, is a derived type with a more limited range than the parent type so it has all of the characteristics of the parent type except that it cannot be assigned a value outside of its defined range.

HOW TO USE SOME OF THE NEW TYPES

As an example of using the new types, consider the type **SALAD_INT** which is a derived type of the parent type **INTEGER**. Since three variables, **Salad**, **Lettuce**, and **Tomatoes**, are all of the same type, they can be freely added, compared, assigned to each other, or used in any way legal for integer class variables. They can be used as the range limits in a **for** loop. The three variables have the same range as the parent type **INTEGER**, namely -2,147,483,648 to 2,147,483,647 on most 32 bit machines. Constants of type universal_integer can be added to, compared to, or assigned to these three variables.

HOW CAN THESE NEW TYPES HELP IN A PROGRAM?

Everything that was said about type **SALAD_INT** in the last paragraph is true of type **ANIMAL_INT**, **TREE_INT**, or any of the other four types. Suppose somewhere in our program we tried to add the number of **Tomatoes** to the number of **Dogs** and assign the result to **Trees**. If the variable names are meaningful, we would probably not want to do such an operation in any practical program. The Ada compiler would give us a compile time error, so we would detect the error before we tried to run the program with such a silly statement. Careful assignment of types can be used to protect us from the silly little errors that we are all so prone to make. It would be much more efficient to let the compiler find these silly little errors and free us up to find the analysis errors we also make.

HOW DO YOU DO A TYPE TRANSFORMATION?

Even though you set things up very carefully, you may need to perform some operations on the data where you actually do need to add the number of **Animals** to the total of **Oak** plus **Coconut**. Line 30 illustrates how to do this. Enclosing the variable in parentheses and adding the desired type to the front of the grouping will change the type from the variable's actual type to the type in front of the parentheses, but only for that one place in the program.

In line 31, **Trees** and **Salad** are both transformed to type **INTEGER** before being summed and assigned to **Count**, a variable of type **INTEGER**. Line 33 illustrates the addition of many variables by first transforming each to type **SALAD_INT** then performing the addition. In line 40, the same variables are added together, but in this case, all variables are transformed into type **ANIMAL_INT** prior to summing, then the sum is transformed to type **SALAD_INT**. The two methods should result in the same answer, and you can verify that they do when you compile and run the program.

WHAT IS A SUBTYPE?

A **subtype** is a new type based on a parent type but usually with a more restricted range. It inherits all of the characteristics of the parent type and in addition, it can be freely intermixed with the parent type in calculations and assignment statements. The reason for using a subtype is usually to declare a variable of the parent type but with a more limited range to take advantage of the range checking capability of Ada.

WE CAN ALSO HAVE SUBTYPES OF DERIVED TYPES

Example program ------> e_c07_p2.ada

```
                                       -- Chapter 7 - Program 2
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure DerSubs is

   type NEW_INT      is new INTEGER range 12..127;
   type NEW_INT_TYPE is new INTEGER;

   subtype SUB_INT     is NEW_INT;
   subtype NEW_SUBTYPE is NEW_INT range 12..127;
   type    DER_SUB     is new NEW_SUBTYPE range 12..32;

   Arrow, Dart : NEW_INT;
   Size        : NEW_INT_TYPE;
   Thing       : INTEGER := 15;
   Point       : NEW_SUBTYPE;

begin

   Size := 10;
   Arrow := 23;
   Dart := 2 * Arrow - 25;
   Dart := Arrow + 2 * (NEW_INT(Size + 2) + NEW_INT(Thing));
   Dart := Arrow + 2 * NEW_INT(Size + NEW_INT_TYPE(Thing));
   Point := Arrow + Dart;

end DerSubs;




-- Result of execution

--   (No output from this program.)
```

The program named e_c07_p2.ada gives an example of the definition and use of a subtype of a derived type, and a derived type of a subtype. In line 7 we declare a derived type and in line 10 we declare a subtype of the new derived type. The subtype of the derived type has the same characteristics as the derived type except that it has a more restricted range in this case. Variables of type **NEW_SUBTYPE** are compatible with variables of their parent type **NEW_INT**. This is illustrated in line 26.

In this case, **NEW_SUBTYPE** is as different from the type **INTEGER**, as **SALAD_INT** was in the last program.

WE CAN HAVE A DERIVED TYPE OF A SUBTYPE

Line 12 illustrates the declaration of a derived type based on using a subtype for the parent type. Note that the new derived type has all of the characteristics of its parent type except for the more restricted range, but once again, it is an entirely new type as far as type checking is concerned.

A SUBTYPE CAN BE SIMPLY A SYNONYM

Line 10 illustrates a subtype which covers the entire range of its parent type. Since variables of this subtype can be freely intermixed with variables of its parent type, the subtype name is simply a synonym for the parent type name.

With the discussion of the last program fresh in your mind, you should breeze through the remainder of this program. Be sure to compile and execute it.

USING OTHER PREDEFINED TYPES FOR THE PARENT

Example program ------> **e_c07_p3.ada**

```
                                                -- Chapter 7 - Program 3
with Ada.Text_IO;
use Ada.Text_IO;

procedure MoreDers is

                                  -- Some floating point types
   type    NEW_FLOAT1 is digits 5;
   type    NEW_FLOAT2 is digits 5 range 1.0..12.0;
   type    DER_FLOAT  is new FLOAT;
   type    LIM_FLOAT  is new FLOAT range 0.0..555.5;
   subtype SUB_FLOAT  is     DER_FLOAT range -2.3..12.8;

                                  -- Some fixed point types
   type    NEW_FIXED1 is delta 0.5  range 1.0..12.0;
   type    NEW_FIXED2 is delta 0.05 range 1.0..12.0;
   type    DER_FIXED  is new NEW_FIXED1;
   type    LIM_FIXED  is new NEW_FIXED1 range 1.0..5.5;
   subtype SUB_FIXED  is     DER_FIXED range 2.1..2.8;

                                  -- Some CHARACTER types
   type    DER_CHAR   is new CHARACTER;
   type    ALPHA_CHAR is new CHARACTER range 'A'..'Z';
   subtype HEX_CHAR   is     ALPHA_CHAR range 'A'..'F';

                                  -- Some enumerated types
   type    DAY         is (MON, TUE, WED, THU, FRI, SAT, SUN);
   type    WEEKDAY     is new DAY range MON..FRI;
   subtype BOWLING_DAY is WEEKDAY range WED..THU;

                                  -- Some floating point objects
   Direction : FLOAT;
   Speed     : DER_FLOAT := 100.0;
   Length    : LIM_FLOAT := 72.41;
   Size      : SUB_FLOAT := 4.3;
```

```
begin

   Direction := 1.2 + FLOAT(Length + LIM_FLOAT(Speed * Size));

end MoreDers;


-- Result of execution

-- (There is no output from this program)
```

Examine the program named e_c07_p3.ada for examples of derived types and subtypes based on some of the other predefined types in Ada. We begin by declaring two user defined types in lines 8 and 9 which are of the floating point class of types because of the reserved word **digits** appearing in the definition. In line 10 we declare **DER_FLOAT** which has all of the characteristics of the predefined type **FLOAT**, except that the compiler will consider it to be an entirely different type and will not allow mixing of these two types. Of course, type conversion can be used if necessary.

The derived type **LIM_FLOAT** is declared with all the characteristics of **FLOAT** except that it has a limited range to allow for compiler checks. Line 12 contains the definition of a subtype based on **DER_FLOAT** with a more limited range. Variables of the type **SUB_FLOAT** can be freely intermixed with variables of type **DER_FLOAT,** but not with those declared with the types **FLOAT**, **NEW_FLOAT1**, **NEW_FLOAT2**, or **LIM_FLOAT**.

Lines 15 through 19 illustrate the same principles applied to fixed point types and should be self explanatory. The only difference is the use of the reserved word **delta** in the fixed point definitions.

Lines 22 through 29 illustrate the declaration of derived types and subtypes of the **CHARACTER** type and an enumerated type. These statements will be left to the students study since they are so similar to the example using **FLOAT** as the parent type. We will study the **CHARACTER** type in chapter 11.

A few variables are declared and some are initialized in lines 32 through 35, and a nonsense calculation is given in line 39 to illustrate the type transformations that can be done with derived types.

Be sure to compile and execute this program even though it has no output.

WHAT IS A CLASS IN ADA?

A class in Ada is the entire group of various concrete types that descend from a common ancestor, or a common root. The name of the root is used as the name of the class. Therefore in the program named e_c07_p3.ada, the **NEW_FIXED1** class consists of the types **NEW_FIXED1**, **DER_FIXED**, **LIM_FIXED**, and **SUB_FIXED**, because the last three are somehow derived from **NEW_FIXED1**. In a similar manner, the class named **DAY** consists of the types **DAY**, **WEEKDAY**, and **BOWLING_DAY**.

A WORD OF SUMMARY ABOUT TYPES

We have seen that in addition to the predefined types, we can declare additional types for use in our programs. We can then use any of the predefined or user defined types as the parent type for either subtypes or derived types. The new subtype or derived type can be used as a parent type for additional subtypes or derived types and we find that we have a tremendous amount of flexibility in defining the data to solve any particular problem.

PROGRAMMING EXERCISE

1. Modify the program named e_c07_p1.ada to include a new instantiation of the package **Ada.Text_IO.Integer_IO** to output the variable named **Salad** in line 37 and 44 without the

type conversion.(Solution)

```
                        -- Chapter 7 - Programming exercise 1
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure CH07_1 is

   type LITTLE_INT is range -24..17;
   type TINY_INT   is range -3..2;
   type POS_INT    is range 25..38;
   type TINY_POS   is new POS_INT range 25..30;
   type SALAD_INT  is new INTEGER;
   type ANIMAL_INT is new INTEGER;
   type TREE_INT   is new INTEGER range -557..1098;

   package Salad_Int_IO is new Ada.Text_IO.Integer_IO(SALAD_INT);
   use Salad_Int_IO;

   Salad    : SALAD_INT;
   Lettuce  : SALAD_INT := 22;
   Tomatoes : SALAD_INT := 14;
   Animals  : ANIMAL_INT;
   Dogs     : ANIMAL_INT := 3;
   Cats     : ANIMAL_INT := 4;
   Trees    : TREE_INT;
   Oak      : TREE_INT := 12;
   Coconut  : TREE_INT := 8;
   Count    : INTEGER;

begin

   Salad := Lettuce + Tomatoes;
   Animals := Dogs + Cats;
   Trees := Oak + Coconut + TREE_INT(Animals);
   Count := INTEGER(Trees) + INTEGER(Salad);

   Salad := SALAD_INT(Dogs) * Tomatoes +
            SALAD_INT(Cats) * SALAD_INT(Oak) +
            SALAD_INT(Count);
   Put("The 1st Salad calculation is ");
   Put(Salad);
   New_Line;

   Salad := SALAD_INT(Dogs * ANIMAL_INT(Tomatoes) +
                      Cats * ANIMAL_INT(Oak) +
                             ANIMAL_INT(Count));
   Put("The 2nd Salad calculation is ");
   Put(Salad);
   New_Line;

end CH07_1;




-- Result of execution

-- The 1st Salad calculation is    153
-- The 2nd Salad calculation is    153
```

Ada Tutorial - Chapter 8

SUBPROGRAMS

LET'S LOOK AT A PROCEDURE

Example program ------> **e_c08_p1.ada**

```
                                                       -- Chapter 8 - Program 1
with Ada.Text_IO;
use Ada.Text_IO;

procedure Proced1 is

      procedure Write_A_Line is
      begin
         Put("This is a line of text.");
         New_Line;
      end Write_A_Line;

begin
   Write_A_Line;
   Write_A_Line;
end Proced1;




-- Result of execution

-- This is a line of text.
-- This is a line of text.
```

Ada was designed to be very modular, and we come to the point where we will study the first and simplest form of modularization, the procedure. If you examine the program named e_c08_p1.ada, you will have your first example of a procedure. Some languages call this kind of subprogram a subroutine, others a function, but Ada calls it a procedure.

THE PROCEDURE IS LIKE THE MAIN PROGRAM

Beginning with the executable part of the main program, we have two executable statements in lines 14 and 15, each calling a procedure to write a line, if the name has any meaning. As always, the two statements are executed in succession, and each statement is a procedure call to which we would like to transfer control. The procedure itself is defined in lines 7 through 11, and a close inspection will reveal that the structure of the procedure is very similar to the structure for the main program. It is in fact, identical to the main program, and just as we begin executing the main program by mentioning its name, we begin executing the procedure by mentioning its name. The procedure has a declarative part, which is empty in this case, and an executable part which says to display a line of text and return the cursor to the beginning of the next line.

When execution reaches the **end** statement of the procedure, the procedure is complete and control returns to the next successive statement in the calling program.

Everything we have said about the main program, which is actually a procedure, is true for the procedure. Thus we can define new types, declare variables and constants, and even define additional procedures in the declarative part of this procedure. Likewise, we can call other procedures, and do assignments and compares in the executable part of the procedure.

ORDER OF DECLARATIONS

The original Ada specification, Ada 83, required that all procedures must come after all type, constant, and variable declarations. This was to force you to arrange your program such that all of the smaller declarations must come first, followed by the larger ones, so that the smaller declarations would not get lost by being buried between the larger declarations. This has been removed from the requirement for Ada 95, and you are permitted to list the various entities in any order you desire.

The embedded procedure has all of the flexibility and requirements as those defined for the main procedure. We will give further examples of each of these points in the next few example programs. At this time, compile and run this program, and make sure you understand its operation completely.

THREE PROCEDURES

Example program ------> **e_c08_p2.ada**

```
                                       -- Chapter 8 - Program 2
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure Proced2 is

   Counter : INTEGER;

   procedure Write_A_Header is
   begin
      Counter := 1;
      Put("This is the heading for this little program.");
      New_Line(2);
   end Write_A_Header;

   procedure Write_And_Increment is
   begin
      Put("This is line number");
      Put(Counter, 2);
      Put_Line(" of this program.");
      Counter := Counter + 1;
   end Write_And_Increment;

   procedure Write_An_Ending_Statement is
   begin
      New_Line;
      Put_Line("This is the end of this little program.");
   end Write_An_Ending_Statement;

begin
   Write_A_Header;
   for Index in 1..7 loop
      Write_And_Increment;
   end loop;
   Write_An_Ending_Statement;
end Proced2;




-- Result of execution

-- This is the heading for this little program.
--
-- This is line number 1 of this program.
-- This is line number 2 of this program.
-- This is line number 3 of this program.
-- This is line number 4 of this program.
-- This is line number 5 of this program.
-- This is line number 6 of this program.
-- This is line number 7 of this program.
--
-- This is the end of this little program.
```

Examine the program named e_c08_p2.ada, and you will see three procedures in the declarative part of this program. Jumping ahead to the main program, beginning in line 30, you will see that the program will write a header, write and increment something 7 times, then write an ending statement. Notice how the use of descriptive procedure names resulted in our understanding of what the program will do without looking at the procedures themselves. The names are long, and a bit tedious to type in, but since Ada was designed to be a language that would be written once and read many times, the extra time will pay off when the source code is studied by several persons in the future.

WHAT IS A GLOBAL VARIABLE?

The variable named **Counter**, defined in line 7, is a global variable because it is defined prior to any procedure. It is therefore available for use by any of these procedures and it is, in fact, used by two of them. In line 11, the variable **Counter** is assigned the value of 1 when the procedure named **Write_A_Header** is called. Each time **Write_And_Increment** is called, the current value of **Counter** is written to the display, along with a message, and the value is incremented. Note carefully that the procedures are not executed in the order they are because of their relative order in the declaration part of the program, but because the main program calls them in that order. The program would work exactly the same way if you moved the first procedure, in its entirety of course, after the procedure **Write_An_Ending_Statement**. The order of execution is controlled by the order of calls, not the physical order of the procedures.

WHY DO PROCEDURES GO IN THE DECLARATIVE PART?

The declarative part of the program is where we define entities for use within the executable part. The procedures are actually definitions of how to do something, so they are right where they belong. Compile and run this program and be sure you understand the output.

A PROCEDURE WITH PARAMETERS

Example program ------> e_c08_p3.ada

```
                                       -- Chapter 8 - Program 3
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure Proced3 is

   Dogs, Cats, Animals : INTEGER;

                          -- This is a procedure specification
   procedure Total_Number_Of_Animals(Variety1 : in     INTEGER;
                                     Variety2 : in     INTEGER;
                                     Total    :    out INTEGER);

                          -- This is a procedure body
   procedure Total_Number_Of_Animals(Variety1 : in     INTEGER;
                                     Variety2 : in     INTEGER;
                                     Total    :    out INTEGER) is
   begin
      Total := Variety1 + Variety2;
   end Total_Number_Of_Animals;
```

```
begin
   Dogs := 3;
   Cats := 4;
   Total_Number_Of_Animals(Dogs, Cats, Animals);
   Put("The total number of animals is");
   Put(Animals, 3);
   New_Line;
end Proced3;




-- Result of execution

-- The total number of animals is  7
```

Examine the file named e_c08_p3.ada and you will find a procedure that requires some data to be supplied to it each time it is called. Three variables are defined as type **INTEGER** and used in the procedure call in line 25. We will ignore the strange looking constructs in lines 9 through 12 for a couple of paragraphs. The procedure header, beginning in line 15, states that it is expecting three parameters, and each must be of type **INTEGER**, so we are compatible so far. When the procedure is called, the value of the first variable, which is named **Dogs** in this case, is taken to the procedure, where the procedure prefers to refer to it by the name **Variety1**. In like manner, the value of **Cats** is given to the procedure, and is called **Variety2**. The variable named **Animals** is referred to by the name **Total** in the procedure. The procedure is now ready to do some meaningful work with these variables, but is somewhat limited in what it can do to them because of the mode field of the procedure header.

THE MODE OF A PARAMETER

The formal parameter, as it is called in the procedure header, named **Variety1**, is of mode **in** which means that it is an input parameter, and therefore cannot be changed within the procedure. **Variety1**, along with **Variety2** for the same reason, is a constant within the procedure, and any attempt to assign a value to it will result in a compile error. The formal parameter named **Total**, however, is of mode **out** and can therefore have a new value assigned to it within the procedure. Ada 83 defined that the **out** mode variable could not be read, but Ada 95 has relaxed this requirement and permits you to read the **out** mode variable within the procedure. Extra care must be taken that you do not attempt to read the value of an **out** mode parameter that has not yet been initialized to some meaningful value. Parameters that are of the modes **in** or **in out** do not have this problem since they are initialized, by definition, when the procedure is called.

If a parameter is defined as being of mode **in out**, it can be both read from and written to. If no mode is given, the system will use mode **in** as a default.

PARAMETER MODE SELECTION

All variables could be defined as mode **in out** and there would be no problems, since there would be maximum flexibility, or so it would seem. There is another rule that must be considered, and that rule says that every parameter that is of mode **out** or **in out** must be called with a variable as the actual parameter in the calling program. This permits the value to be returned and assigned to the variable. A variable of mode **in** can use a constant or a variable for the actual parameter in the calling program. We have been using the **New_Line** procedure with a constant in it, such as **New_Line(2)**, and if it had been defined with the formal parameter of mode **in out**, we would have had to define a variable, assign the value 2 to it, and use the variable in the call. This would have made the procedure a bit more difficult to use, and in fact, somewhat awkward. For this reason, the formal parameter in **New_Line** was defined using mode **in**. You should choose the mode of the formal parameters very carefully.

The three formal parameters are available for use in the procedure, once they are defined as illustrated, just as if they had been defined in the declarative portion of the procedure. They are not available to any other procedure or the main program, because they are defined locally for the procedure. When used however, their use must be consistent with the defined mode for each.

SOME GLOBAL VARIABLES

The three variables declared in line 7 can be referred to in the procedure as well as in the main program. Because it is possible to refer to them in the procedure, they can be changed directly within the procedure. The variable **Animals** can also be modified when control returns to the main program because it is declared as **out** mode in the procedure header. This possibility can lead to some rather unusual results. You should spend some time thinking about what this really means.

THE PROCEDURE SPECIFICATION

Lines 10, 11, and 12, give an example of a procedure specification. This is an incomplete procedure declaration that you will find useful when you begin writing larger programs. The procedure specification can be included for any procedure, if desired, and it describes the external interface to the procedure without declaring what the procedure actually does. The Pascal programmer will recognize this as being very similar to the forward declaration. More will be said about this topic later. Note that the procedure specification is not required in this case, it is only included as an illustration.

Compile and run this program after you understand the simple addition and assignment that is done for purposes of illustration.

PROCEDURES CALLING OTHER PROCEDURES

Example program ------> **e_c08_p4.ada**

```
                                       -- Chapter 8 - Program 4
with Ada.Text_IO;
use Ada.Text_IO;

procedure Calling is

   procedure One is
   begin
      Put("This is procedure One talking.");
      New_Line;
   end One;

   procedure Two is
   begin
      Put("This is procedure Two talking.");
      New_Line;
      One;
   end Two;

   procedure Three is
   begin
      Put("This is procedure Three talking.");
      New_Line;
      Two;
      One;
   end Three;

begin
   One;
   Two;
   Three;
end Calling;
```

```
-- Result of execution

-- This is procedure One talking.
-- This is procedure Two talking.
-- This is procedure One talking.
-- This is procedure Three talking.
-- This is procedure Two talking.
-- This is procedure One talking.
-- This is procedure One talking.
```

The example program e_c08_p4.ada contains examples of a procedure calling another procedure, which is perfectly legal if the called procedure is within the scope of the calling procedure. Much more will be said about scope later in this tutorial. The only rule that will be mentioned here is that the procedure must be defined prior to a call to it. You should have no trouble understanding this program, and when you do, you should compile and execute it. Be sure you understand where each line in the output comes from and why it is listed in the order that it is.

HOW DO WE NEST PROCEDURES?

Example program ------> **e_c08_p5.ada**

```
                                        -- Chapter 8 - Program 5
with Ada.Text_IO;
use Ada.Text_IO;

procedure Nesting is

   procedure Triple is

         procedure Second_Layer is

               procedure Bottom_Layer is
               begin
                  Put_Line("This is the Bottom Layer talking.");
               end Bottom_Layer;

         begin
            Put_Line("This is the Second Layer talking.");
            Bottom_Layer;
            Put_Line("We are back up to the Second Layer.");
         end Second_Layer;

   begin
      Put_Line("This is procedure Triple talking to you.");
      Second_Layer;
      Put_Line("We are back up to the procedure named Triple.");
   end Triple;

begin
   Put_Line("Start the triple nesting here.");
   Triple;
   Put_Line("Finished, and back to the top level.");
end Nesting;
```

```
-- Result of execution

-- Start the triple nesting here.
-- This is procedure Triple talking to you.
-- This is the Second Layer talking.
-- This is the Bottom Layer talking.
-- We are back up to the Second Layer.
-- We are back up to the procedure named Triple.
-- Finished, and back to the top level.
```

Examine the program named e_c08_p5.ada for examples of nested procedures. We mentioned earlier that it was possible to embed a procedure within the declarative part of any other procedure. This is illustrated in lines 9 through 20 where the procedure **Second_Layer** is embedded within the procedure **Triple**. In addition, the procedure **Second_Layer** has the procedure **Bottom_Layer** embedded within its declarative part in lines 11 through 14. Such nesting can continue indefinitely, because there is no limit to the depth of nesting allowed in Ada.

VISIBILITY OF PROCEDURES

There is a limit on visibility of procedures. Any procedure is visible, and can therefore be called, if it is within the top level of the declarative part of the calling procedure. Any procedure is also visible if it is prior to, on the same level, and within the same declarative part as the calling point. Finally, any procedure can be called if it is prior to, on the same level, and within the same declarative part as any subprogram within which the calling point is nested. In simpler words, a procedure is visible in three cases. First, if it is within the declarative part of the calling procedure. The second case is if it is a peer (on the same level within a parent subprogram) or thirdly, if it is a peer of any parent.

The procedure named **Triple** can therefore call **Second_Layer**, but not **Bottom_Layer**, since it is at a lower level and is not visible. The main program, according to these rules, is only allowed to call **Triple**, because the other two procedures are nested too deeply for a direct call. Be sure to compile and run this program and study the results.

ADA FUNCTIONS

Example program ------> **e_c08_p6.ada**

```
                                       -- Chapter 8 - Program 6
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure Funct is

   Twelve : INTEGER := 12;
   Sum    : INTEGER;
                            -- This is a function specification
   function Square(Val : INTEGER) return INTEGER;

                            -- This is a function body
   function Square(Val : INTEGER) return INTEGER is
   begin
      return Val * Val;
   end Square;

   function Sum_Of_Numbers(Val1, Val2 : INTEGER) return INTEGER is
   begin
      return Val1 + Val2;
   end Sum_Of_Numbers;

begin
```

```
   Put("The square of 12 is");
   Put(Square(Twelve), 4);
   New_line;
   Sum := Sum_Of_Numbers(Twelve, 12);
   Put("The sum of 12 and 12 is");
   Put(Sum, 4);
   New_Line;
end Funct;




-- Result of execution

-- The square of 12 is 144
-- The sum of 12 and 12 is  24
```

The program named e_c08_p6.ada has two examples of Ada functions. A function differs from a procedure in only two ways. A function returns a single value which is used in the place of its call, and all formal parameters of a function must be of type **in**, with no other mode permitted. In the program under consideration, two functions are illustrated, one beginning in line 13, and the other beginning in line 18. Note that each begins with the reserved word **function**.

A FUNCTION SPECIFICATION

In a manner similar to that defined for a procedure we can define a function specification that gives the interface of the function to any potential caller. You will find the function specification useful later in your Ada programming efforts. It is similar to the Pascal forward declaration. Note once again, that the function specification is not required in this case, it is only given here as an illustration.

The function named **Square** requires one argument which it prefers to call **Val**, and which must be of type **INTEGER**. It returns a value to the main program which will be of type **INTEGER** because that is the type given between the reserved words **return** and **is** in the function header in line 13.

A function must return a value, and the value is returned by following the reserved word **return** with the value to be returned. This return must be done in the executable part of the program, as illustrated in line 15. It is an error to fail to execute a **return** statement and fall through the end of a function. Such a runtime error will be reported by raising the exception **Program_Error**, which will be explained later in this tutorial.

CAN YOU RETURN FROM A PROCEDURE?

It would be well to point out that you can return from a procedure by using the **return** statement also, but no value can be given since a procedure does not return a value in the same manner as a function. The return statement can be anyplace in the procedure or function and there can be multiple returns if the logic dictates the possibility of returning from several different places.

A VALUE IS SUBSTITUTED FOR THE FUNCTION CALL

Examining the executable part of the program, we find that the variable **Twelve** has been initialized to the value of 12, and is used in line 25 as the argument for the function **Square**. This causes **Square** to be called, where the value of 12 is squared and the result is returned as 144. It is as if the resulting value of 144 replaces the function call **Square(Twelve)** in the **Put** procedure call, and the value of 144 is displayed. Continuing on to line 27, the variable **Twelve**, which still contains the value of 12, and the constant 12, are given to the function **Sum_Of_Numbers** which returns the sum of 24. This value is assigned to the variable named **Sum** where it is stored for use in line 29.

A function can be defined with no input parameters, in which case, the function is called with no parameters. Such a case would be useful for a random number generator where the call could be X := Random; assuming a new random number is returned each time the function is called. Compile and execute this program and study the output generated.

A FULLER EXAMPLE

Example program ------> e_c08_p7.ada

```
                                       -- Chapter 8 - Program 7
-- This is a program to return an odd square of a number.  The result
--   will have the sign of the original number, and the magnitude of
--   the square of the original number.  The entire program can be
--   written in one line (i.e. - Result := Index * abs(Index); ), but
--   the purpose of the program is to illustrate how to put various
--   procedures and functions together to build a useable program.
-- Various levels of nesting are illustrated along with several
--   parameters being used in procedure calls.

with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure OddSqre is

   Result : INTEGER;

   function Square_The_Number(Number_To_Square : in INTEGER)
                                                return INTEGER is
   begin
      return Number_To_Square * Number_To_Square;
   end Square_The_Number;

   procedure Square_And_Keep_Sign(Input_Value   : in INTEGER;
                                  Funny_Result : out INTEGER) is

        procedure Do_A_Negative_Number(InVal   : in INTEGER;
                                       OutVal : out INTEGER) is
        begin
           OutVal := -Square_The_Number(InVal);
        end Do_A_Negative_Number;

   begin
      if Input_Value < 0 then
         Do_A_Negative_Number(Input_Value, Funny_Result);
         return;
      elsif Input_Value = 0 then
         Funny_Result := 0;
         return;
      else
         Funny_Result := Square_The_Number(Input_Value);
         return;
      end if;
   end Square_And_Keep_Sign;

begin

   for Index in -3..4 loop
      Square_And_Keep_Sign(Index, Result);
      Put("The Odd Square of");
      Put(Index, 3);
      Put(" is");
      Put(Result, 5);
      New_Line;
   end loop;

end OddSqre;




-- Result of execution

-- The Odd Square of -3 is   -9
-- The Odd Square of -2 is   -4
-- The Odd Square of -1 is   -1
-- The Odd Square of  0 is    0
-- The Odd Square of  1 is    1
-- The Odd Square of  2 is    4
-- The Odd Square of  3 is    9
-- The Odd Square of  4 is   16
```

Examine the program named e_c08_p7.ada which is a rather odd program that computes the square of an integer type variable, but maintains the sign of the variable. In this program, the odd square of 3 is 9, and the odd square of -3 is -9. Its real purpose is to illustrate several procedures and a function interacting.

The main program named **OddSqre** has a function and a procedure nested within its declarative part, both of which have parameters passed. The nested procedure named **Square_And_Keep_Sign** has another procedure nested within its declarative part, named **Do_A_Negative_Number** which calls the function declared at the next higher level.

This program is a terrible example of how to solve the problem at hand but is an excellent example of several interacting subprograms, and it would be profitable for you to spend enough time with it to thoroughly understand what it does.

COMMENTS ON e_c08_p7.ada

This program illustrates some of the options that are purely programming taste. The first option is illustrated in line 21, where we could have chosen to use the construct **Number_To_Square**2** instead of the simple multiplication. Either form is correct and the one to be used should reflect the nature of the problem at hand. The second option is the fact that three returns were included in lines 36, 39, and 42, when a single return could have been used following the end of the **if** statement. This was done to illustrate multiple returns in use. In some cases, the logic of the program is much clearer to use several returns instead of only one. More than anything else, it is a matter of personal taste. Be sure to compile and execute this program.

OVERLOADING

Example program ------> **e_c08_p8.ada**

```
                                       -- Chapter 8 - Program 8
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure Overload is

   Int_Dat : INTEGER;
   Flt_Dat : FLOAT;


   function Raise_To_Power(Index : INTEGER) return INTEGER is
   begin
```

```
      Put_Line("In the INTEGER function.");
      return Index * Index;
   end Raise_To_Power;



   function Raise_To_Power(Value : FLOAT) return FLOAT is
   begin
      Put_Line("In the FLOAT function.");
      return Value * Value * Value;
   end Raise_To_Power;



   procedure Raise_To_Power(Index  : in     INTEGER;
                            Result :    out INTEGER) is
   begin
      Put_Line("In the INTEGER procedure.");
      Result := Index * Index * Index;
   end Raise_To_Power;



   procedure Raise_To_Power(Value  : in     FLOAT;
                            Result :    out FLOAT) is
   begin
      Put_Line("In the FLOAT procedure.");
      Result := Value * Value;
   end Raise_To_Power;

begin

   Int_Dat := Raise_To_Power(2);     -- uses INTEGER function

   Flt_Dat := Raise_To_Power(3.2);   -- uses FLOAT function

   Raise_To_Power(3, Int_Dat);       -- uses INTEGER procedure

   Raise_To_Power(2.73, Flt_Dat);    -- uses FLOAT procedure

   Int_Dat := 2;
                      -- In the following statement,
                      -- the function returns 2 squared, or 4
                      -- and the procedure cubes it to 64.
   Raise_To_Power(Raise_To_Power(Int_Dat), Int_Dat);
   Put("The result is ");
   Put(Int_Dat, 4);
   New_Line;

end Overload;




-- Result of execution

-- In the INTEGER function.
-- In the FLOAT function.
-- In the INTEGER procedure.
-- In the FLOAT procedure.
-- In the INTEGER function.
-- In the INTEGER procedure.
-- The result is   64
```

We have casually mentioned overloading earlier in this tutorial and it is now time to get a good example of what overloading is by examining the program named e_c08_p8.ada. This program includes two functions and two procedures and all four of these subprograms have the same name. The Ada system has the ability to discern which subprogram you wish to use by the types included in the actual parameter list and the type of the return. In line 45, we make a call to a function with a 2, which is an integer type constant, and we assign the returned value to an **INTEGER** type variable. The system will look for a function named **Raise_To_Power** with a single integer class formal parameter and an **INTEGER** type return which it finds in lines 11 through 15, so it executes this function. The actual searching will be done at compile time so the efficiency is not degraded in any way by the overloaded names.

If you continue studying this program you will see how the system can find the correct subprogram by comparing types used as formal parameters, and the type returned. Using the same name for several uses is referred to as overloading the subprogram names and is an important concept in the Ada language.

OVERLOADING CAN CAUSE YOU PROBLEMS

If we made an error in this example program, by inadvertently omitting the decimal point in line 47, and assigning the result to an **INTEGER** type variable, the system would simply use the wrong function and generate invalid data for us. An even worse problem could be found if we had a function that used an **INTEGER** for input and a **FLOAT** for output, because only one small error could cause erroneous results. Because of this, it would be to your advantage to use different subprogram names for different operations, unless using the same name results in clear code.

In the case of the text output procedures which we have been using, it makes a lot of sense to overload the output subprograms to avoid confusion. The name **Put** is used for outputting strings, integers, enumerated types, etc, and we are not confused. Overloading can be an advantage in certain cases but should not be abused just because it is available. Be sure to compile and execute this program.

HOW ARE PARAMETERS PASSED TO A SUBPROGRAM?

If you understand the method of passing parameters to and from a subprogram, it may occasionally be possible to improve the efficiency by selecting the type carefully. For that reason the following, admittedly sketchy, descriptions are given;

- Parameters of scalar types are always passed by copy, and when the subprogram returns, the new value is copied back to the original. Therefore, in the case of **out** or **in out** parameters, intermediate values are not reflected back to the original until the subprogram is complete.
- Composite parameters (array and record) can use either a copy as defined above, or a reference to the original. The compiler writer has the option of choosing the means of parameter passing in this case.
- The more advanced types, such as task types or protected types, are always passed by a reference to the original.

Generally, it shouldn't matter to you how the various types are passed, but it might matter in some situations.

PROGRAMMING EXERCISES

1. Rewrite e_c05_p6.ada to do the temperature conversion in a procedure.(Solution)

-- Chapter 8 - Programming exercixe 1

```
-- Centigrade to Farenheit temperature table
--
```

```
--  This program generates a list of Centigrade and Farenheit
--    temperatures with a note at the freezing point of water
--    and another note at the boiling point of water.

with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure CH08_1 is

   Centigrade, Farenheit : INTEGER;

   procedure Cent_To_Faren(Cent  : in     INTEGER;
                           Faren :    out INTEGER) is
   begin
      Faren := 32 + Cent * 9 / 5;
   end Cent_To_Faren;

begin
   Put("Centigrade to Farenheit temperature table");
   New_Line(2);
   for Count in INTEGER range -2..12 loop
      Centigrade := 10 * Count;
      Cent_To_Faren(Centigrade,Farenheit);
      Put("C =");
      Put(Centigrade,5);
      Put("    F =");
      Put(Farenheit,5);
      if Centigrade = 0 then
         Put("  Freezing point of water");
      end if;
      if Centigrade = 100 then
         Put("  Boiling point of water");
      end if;
      New_Line;
   end loop;
end CH08_1;




-- Result of execution

-- Centigrade to Farenheit temperature table
--
-- C =  -20    F =   -4
-- C =  -10    F =   14
-- C =    0    F =   32  Freezing point of water
-- C =   10    F =   50
-- C =   20    F =   68
-- C =   30    F =   86
-- C =   40    F =  104
-- C =   50    F =  122
-- C =   60    F =  140
-- C =   70    F =  158
-- C =   80    F =  176
-- C =   90    F =  194
-- C =  100    F =  212  Boiling point of water
-- C =  110    F =  230
-- C =  120    F =  248
```

2. Rewrite e_c05_p6.ada to do the temperature conversion in a function.(Solution)

-- Chapter 8 - Programming exercise 2

```
-- Centigrade to Farenheit temperature table
--
--  This program generates a list of Centigrade and Farenheit
--    temperatures with a note at the freezing point of water
--    and another note at the boiling point of water.

with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure CH08_2 is

   Centigrade, Farenheit : INTEGER;

   function Cent_To_Faren(Cent : INTEGER) return INTEGER is
   begin
      return (32 + Cent * 9 / 5);
   end Cent_To_Faren;

begin
   Put("Centigrade to Farenheit temperature table");
   New_Line(2);
   for Count in INTEGER range -2..12 loop
      Centigrade := 10 * Count;
      Farenheit := Cent_To_Faren(Centigrade);
      Put("C =");
      Put(Centigrade,5);
      Put("    F =");
      Put(Farenheit,5);
      if Centigrade = 0 then
         Put("  Freezing point of water");
      end if;
      if Centigrade = 100 then
         Put("  Boiling point of water");
      end if;
      New_Line;
   end loop;
end CH08_2;




-- Result of execution

-- Centigrade to Farenheit temperature table
--
-- C =  -20    F =   -4
-- C =  -10    F =   14
-- C =    0    F =   32  Freezing point of water
-- C =   10    F =   50
-- C =   20    F =   68
-- C =   30    F =   86
-- C =   40    F =  104
-- C =   50    F =  122
-- C =   60    F =  140
-- C =   70    F =  158
-- C =   80    F =  176
-- C =   90    F =  194
-- C =  100    F =  212  Boiling point of water
-- C =  110    F =  230
-- C =  120    F =  248
```

3. As mentioned in the text, add a function to the program e_c08_p8.ada that uses an **INTEGER** for input and returns a **FLOAT** type result. Remove the decimal point from line 47 to see that the new function is called when we really intended to call the procedure with the floating point number.(Solution)

```
                          -- Chapter 8 - Programming exercise 3
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure CH08_3 is

   Int_Dat : INTEGER;
   Flt_Dat : FLOAT;


   function Raise_To_Power(Index : INTEGER) return INTEGER is
   begin
      Put_Line("In the INTEGER function.");
      return Index * Index;
   end Raise_To_Power;



   function Raise_To_Power(Value : FLOAT) return FLOAT is
   begin
      Put_Line("In the FLOAT function.");
      return Value * Value * Value;
   end Raise_To_Power;



   function Raise_To_Power(Value : INTEGER) return FLOAT is
   begin
      Put_Line("In the new function.");
      return FLOAT(Value * Value * Value);
   end Raise_To_Power;



   procedure Raise_To_Power(Index  : in     INTEGER;
                            Result :    out INTEGER) is
   begin
      Put_Line("In the INTEGER procedure.");
      Result := Index * Index * Index;
   end Raise_To_Power;



   procedure Raise_To_Power(Value  : in     FLOAT;
                            Result :    out FLOAT) is
   begin
      Put_Line("In the FLOAT procedure.");
      Result := Value * Value;
   end Raise_To_Power;

begin

   Int_Dat := Raise_To_Power(2);     -- uses INTEGER function
   Flt_Dat := Raise_To_Power(3);     -- uses new function

   Raise_To_Power(3,Int_Dat);        -- uses INTEGER procedure

   Raise_To_Power(2.73,Flt_Dat);     -- uses FLOAT procedure

   Int_Dat := 2;
                      -- In the following statement,
                      -- the function returns 2 squared, or 4
                      -- and the procedure cubes it to 64.
   Raise_To_Power(Raise_To_Power(Int_Dat),Int_Dat);
   Put("The result is ");
   Put(Int_Dat);
   New_Line;

end CH08_3;




-- Result of execution

-- In the INTEGER function.
-- In the new function.
-- In the INTEGER procedure.
-- In the FLOAT procedure.
-- In the INTEGER function.
-- In the INTEGER procedure.
-- The result is     64
```

Ada Tutorial - Chapter 9

BLOCKS AND SCOPE OF VARIABLES

LARGE PROJECT DECOMPOSITION

Since Ada is a highly structured language, it has the means to divide a large project into many smaller projects through use of procedures and functions. Because procedures and functions can be nested within other procedures and functions, we have the problem of visibility and scope of types, variables, constants, and subprograms.

WHAT IS THE SCOPE OF A VARIABLE?

Example program ------> **e_c09_p1.ada**

```
                                       -- Chapter 9 - Program 1
procedure Scope is

   Count : INTEGER;

   procedure Level1 is
      Index : INTEGER;

      procedure Level2 is
         Count : INTEGER;
      begin
         null;
      end Level2;

      procedure Level2_Prime is
         Data : INTEGER;
      begin
         null;
      end Level2_Prime;

   begin
      null;
   end Level1;

   procedure Other_Level1 is
   begin
      null;
   end Other_Level1;

begin
   null;
end Scope;




-- Result of execution

--   (No output from this program.)
```

Examine the program named e_c09_p1.ada for several examples of variables with different scopes. You should spend a few minutes familiarizing yourself with the structure of the program which contains the main program, or procedure, and four procedures embedded within it. We will begin with the variable named **Count** declared in line 4, and state that it has a scope which extends from the semicolon at the end of its declaration to the end of the entire program in line 32. Its scope extends to the end of the program because it is declared in the declaration part of the main program. It is commonly referred to as a global variable.

WHERE IS A VARIABLE VISIBLE?

The variable named **Count**, declared in line 4, is visible anyplace in the range of its scope, except for one small area of the program. Since another variable with the same name is defined in line 10, the first one is effectively hidden from view within the range of the newer, local variable. Note that it is the local variable that takes precedence and hides the global variable, rather than the other way. The variable named **Count** from line 4, is not visible from the end of line 10 through the end of line 13. It should be clear that the scope of the local variable extends to the end of the executable portion of the subprogram in which it is declared.

In like manner, the variable named **Index**, defined in line 7, has a scope that extends from the end of line 7 to the end of its procedure which ends in line 23. The variable named **Index** is visible throughout its range, because there are no other variables of the same name in a lower level subprogram. The variable named **Data** is visible throughout its range which extends from the end of line 16 through the end of line 19.

THAT WAS ACTUALLY A LIE

The global variable **Count** is not visible from lines 10 through 13, but there is a way to use it in spite of its hidden nature. This will be the topic of the next example program, but you should compile and run the present program to see that it really will compile as given. There is no output, so execution will be uninteresting.

USING THE DOT NOTATION

Example program ------> e_c09_p2.ada

```
                                        -- Chapter 9 - Program 2
procedure Scope2 is

   Count, Index : INTEGER;

   procedure Level1 is
      Index, Count : INTEGER;

      procedure Level2 is
         Count : INTEGER;
      begin
         Count :=                              -- Count from line 10
                  Scope2.Count;                -- Count from line 4
      end Level2;

      procedure Level2_Prime is
         Data, Index, Count : INTEGER;
         Outer_Index : INTEGER renames Scope2.Level1.Index;
      begin

         Count := Index                         -- Count from line 17
                        + Scope2.Level1.Count;  -- Count from line 7

         Index :=                               -- Index from line 17
                  Scope2.Level1.Index +         -- Index from line 7
                   Scope2.Index;                -- Index from line 4

         Index :=                               -- Index from line 17
                  Outer_Index +                 -- Index from line 7
                   Scope2.Index;                -- Index from line 4

      end Level2_Prime;
   begin
      null;
   end Level1;

   procedure Other_Level1 is
   begin
      Count := Index;                              -- Both from line 4
   end Other_Level1;

begin
   Count := Index;                                 -- Both from line 4
end Scope2;




-- Result of execution

-- (No output from this program)
```

Examine the program named e_c09_p2.ada for some examples of making an invisible variable visible. The careful observer will notice that this is the structure of the last program with additional variables declared, and some added assignment statements.

We will consider three variables of the same name, **Count**, and see that we can use all three variables in a single statement if we so desire. Assume we are at line 12 in the program where we wish to use the local variable named **Count**, the one that was declared in line 10. By the definition of Ada, the innermost variable will take precedence and by simply using the name **Count**, we are using the desired one. If however, we would like to use the one declared in line 4, we can do so by using the "dot" notation illustrated in line 13. We are giving the compiler a complete map on where to find the variable. The dot notation can be read as follows, "Go to the outer level, **Scope2**, dot, and the variable named **Count**." Line 13 is therefore referring to the variable declared in line 4. Using the notation **Scope2.Level1.Count** would refer to the variable declared in line 7.

Additional examples of the use of dot notation to use otherwise invisible variables are given in lines 21 through 30. This is also called the expanded name of the variable or the expanded naming convention.

RENAMING A VARIABLE

In order to reduce the number of keystrokes used and to improve the clarity of some programs, Ada provides a renaming capability. Line 18 illustrates this by renaming the triple component combination to the much simpler name, **Outer_Index**. Anyplace in the procedure where it is permissible to use the longer name, it is also legal to use the new shorter name, because they are simply synonyms for the same actual variable. This is a construct that could easily be abused in a program and make a program unnecessarily complicated, so it should be used sparingly.

Compile and run this program, even though it has no output, to assure yourself that it actually will compile. The dot notation will be used in many other places in Ada, so you should become familiar with it.

AN ADA BLOCK

Example program ------> **e_c09_p3.ada**

```
                                       -- Chapter 9 - Program 3
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure Blocks is

   Index, Count : INTEGER;

begin
   Index := 27;
   Count := 33;
   Put("In the main block      - values are");
   Put(Index, 5);                            -- Blocks.Index
   Put(Count, 5);                            -- Blocks.Count
   New_Line;

   declare
      Index, Stuff : INTEGER := -345;
   begin
      Index := 157;
      Put("In the embedded block  - values are");
      Put(Blocks.Index, 5);                  -- Blocks.Index
      Put(Index, 5);                         -- local Index
      Put(Stuff, 5);                         -- local Stuff
      Put(Count, 5);                         -- Blocks.Count
      New_Line;
   end;

   Put("Back to the main block - values are");
   Put(Index, 5);                            -- Blocks.Index
   Put(Count, 5);                            -- Blocks.Count
   New_Line;

   Who:                                      -- Block name
   declare
      Index, Stuff : INTEGER := -345;
   begin
      Index := 157;
      Put("In the block named Who - values are");
      Put(Blocks.Index, 5);                  -- Blocks.Index
      Put(Index, 5);                         -- Who.Index
      Put(Who.Index, 5);                     -- Who.Index
      Put(Stuff, 5);                         -- Who.Stuff
      Put(Who.Stuff, 5);                     -- Who.Stuff
      Put(Count, 5);                         -- Blocks.Count
      New_Line;
   end Who;

   Put("Back to the main block - values are");
   Put(Index, 5);                            -- Blocks.Index
   Put(Count, 5);                            -- Blocks.Count
   New_Line;

end Blocks;




-- Result of execution

-- In the main block      - values are   27   33
-- In the embedded block  - values are   27  157 -345   33
-- Back to the main block - values are   27   33
-- In the block named Who - values are   27  157  157 -345 -345   33
-- Back to the main block - values are   27   33
```

Examine the program named e_c09_p3.ada for an example of the use of an Ada block. Just as you can define a procedure and jump to it, by calling it of course, Ada allows you to define the equivalent of a procedure and execute it as inline code. Such a section of code is called a block and is constructed by using three reserved words, **declare**, **begin**, and **end**, with declarations between the **declare** and **begin**, and executable statements between the **begin** and **end**. Any new types, subtypes, variables, constants, and even subprograms can be declared in the declaration part of the block and used in the executable part. The scope of the declarations begin where they are declared, and end at the end of the block.

A BLOCK IS A SINGLE STATEMENT

A block is a single statement and because it is, it can be put anywhere that it is legal to put any other executable statement. It could be used within a **loop**, in one branch of an **if** statement, or even as one of the cases of a **case** statement. The example program contains two such blocks, the first in lines 17 through 27, and the second in lines 34 through 47. The only real difference is that the second block is a named block, with the name **Who**, the use of which will be defined shortly.

Study the program and you will see that even though there are several variables defined in the block, and at least one is a repeat of a global variable, all are actually available through use of the dot notation defined during our study of the last program. In the first block, the local variables are the default variables when there is a repeated name, but in the second block, the variables can be specifically named by using the dot notation. This is possible because the block is named, the name being **Who** in this particular case. The name is mentioned just prior to the block followed by a colon, and the name is repeated following the **end** of the block. In this case, the name does nothing for you, but if there were two nested blocks, either or both could be named, and you would be able to select which variable you were interested in. There is no limit to the number of blocks that can be nested.

Note that the name used for a block is not a label to which you can jump in order to execute a **goto** statement. The name is used only to name the block.

If no declarations are needed, you can declare a block without the reserved word **declare**, using only the execution block delimiters **begin** and **end**. Without the declaration part, there is little reason to declare the block until we come to the topic of exception handling where it will be extremely useful to have this capability. Compile and execute this program and study the results. Be sure you understand where each of the displayed values come from.

Examine section 5.6 of the Ada 95 Reference Manual (ARM) to gain a little more experience in working with the ARM. You may be quite surprised at the brevity of this section about the block.

WHAT ARE AUTOMATIC VARIABLES?

Example program ------> **e_c09_p4.ada**

```
                                     -- Chapter 9 - Program 4
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure Automatc is

   Dog, Cat : INTEGER;
   Pig, Cow : FLOAT;

begin
```

```
   for Index in 1..10 loop
      Put("The value of Index is");
      Put(Index, 3);

      declare
         START : constant INTEGER := Index;
         STOP  : constant INTEGER := START + 5;
         Count_Stuff : INTEGER;
      begin
         Count_Stuff := START + STOP + Index + 222;
         Put(" --->");
         for Index in START..STOP loop
            Put(Index, 5);
         end loop;
      end;

      New_Line;
   end loop;

end Automatc;




-- Result of execution

-- The value of Index is  1 --->    1    2    3    4    5    6
-- The value of Index is  2 --->    2    3    4    5    6    7
-- The value of Index is  3 --->    3    4    5    6    7    8
-- The value of Index is  4 --->    4    5    6    7    8    9
-- The value of Index is  5 --->    5    6    7    8    9   10
-- The value of Index is  6 --->    6    7    8    9   10   11
-- The value of Index is  7 --->    7    8    9   10   11   12
-- The value of Index is  8 --->    8    9   10   11   12   13
-- The value of Index is  9 --->    9   10   11   12   13   14
-- The value of Index is 10 --->   10   11   12   13   14   15
```

This is a good time to discuss a very important topic that can have a significant effect on how you write some of your programs in the future. The topic is automatic variables, what they are and what they do for you. The best way to define them is to examine another program, and the program named e_c09_p4.ada is written just to illustrate this point.

The program is actually very simple since it is merely one big loop in which the variable **Index** covers the range from 1 through 10. Each time we pass through the loop, the block in lines 16 through 26 is executed and contains another loop to output some integer type data. Take careful notice of the constants and the way they are used, and you will see something that is a little strange. Each time we enter the block, we enter with a larger value for the loop variable, in this case named **Index**, and therefore the constants are different each time through the block. During each pass, however, they are constant, and will be treated as such. This behavior is perfectly normal and will be clearly understood when we define an automatic variable.

Prior to entering the block, the two constants, and the variable named **Count_Stuff** do not exist. When the block is entered, the constants are generated by the system, initialized to their constant values, and available for use within the block. Since the constants are generated each time the block is entered, it is possible to initialize them to a different value each time, which is exactly what is being done here. The process of assigning the constants their values is called elaboration.

The variable is also generated, and made available for use within the block where it is assigned a nonsense value for illustrative purposes, then never used. When program control leaves the block, in this case dropping out the bottom, the two constants and the variable disappear from existence entirely and are no longer available for use anywhere in the program. They are said to have a limited lifetime, their lifetime being from the time they are elaborated until we leave the block in which they are declared. The scope and lifetime of a variable or constant is therefore very important for you to understand.

It should be clear to you that the outer loop variable, **Index**, is visible from lines 12 through 29 except for lines 23 through 25. Within the region of lines 23 through 25, the outer loop variable cannot be used, even by using the expanded naming convention (i.e. - the dot notation), because there is no name for the outer loop. Naming the outer loop would make the outer loop variable available.

WHERE ELSE IS THIS USED?

This concept of the automatic variable is very important because it is used in so many places throughout an Ada program. It is used in four different places in the present example program, once in the block, as we have just mentioned, once in the main program itself, where the four variables with animal names are automatically generated, and twice in the **for** loops. The variables named **Index** in each of the **for** loops are automatic variables that are generated when the loop is entered, and discarded when the loop is completed. As you can see, there is a very good reason why the loop control variable is not available after you leave the loop.

Each time you call a procedure or function, the formal parameters are generated, as are the defined variables and constants. The process of variable generation and constant initialization is called elaboration. They are then used within the subprograms, and discarded when the procedure or function is completed and control is returned to the calling program.

Since the main program is itself a procedure, its variables are handled the same way.

THE STACK STORES THE AUTOMATIC ENTITIES

The generated constants and variables are stored on an internal stack, the definition of which is beyond the scope of this tutorial. If you understand what a stack is, it should be clear to you how the system can generate items, place them on the stack, use them, and discard them. Also, due to the nature of a stack, it should be clear to you how additional variables can be placed on the stack as calls are made to more deeply nested procedures and functions. Finally, it is only because of this use of automatic variables that recursive subprograms can be used. Ada requires that all subprograms be re-entrant and use of the stack makes this possible.

If the last paragraph is too technical for you, don't worry about it, because it is only mentioned for general information, and is not needed to understand Ada programming.

Compile and run e_c09_p4.ada and study the output. Be sure you understand the concept of the automatic variable because some of the advanced programming techniques in Ada will require this knowledge.

PROGRAMMING EXERCISES

1. Modify e_c09_p3.ada to include a procedure in the declaration part of the first block. The procedure should output a line of text to the monitor when called from the executable part of the block. Why can't this procedure be called from outside of the block?(Solution)

```
                            -- Chapter 9 - Programming exercise 1
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure CH09_1 is

   Index, Count : INTEGER;

begin
```

```
   Index := 27;
   Count := 33;
   Put("In the main block      - values are");
   Put(Index, 5);                             -- CH09_1.Index
   Put(Count, 5);                             -- CH09_1.Count
   New_Line;

   declare
      Index, Stuff : INTEGER := -345;

      procedure Output_A_Line is
      begin
         Put_Line("This is in the new block procedure");
      end Output_A_Line;

   begin
      Index := 157;
      Put("In the embedded block  - values are");
      Put(CH09_1.Index, 5);                   -- CH09_1.Index
      Put(Index, 5);                          -- local Index
      Put(Stuff, 5);                          -- local Stuff
      Put(Count, 5);                          -- CH09_1.Count
      New_Line;
      Output_A_Line;
   end;

   Put("Back to the main block - values are");
   Put(Index, 5);                             -- CH09_1.Index
   Put(Count, 5);                             -- CH09_1.Count
   New_Line;

   Who:                                       -- Block name
   declare
      Index, Stuff : INTEGER := -345;
   begin
      Index := 157;
      Put("In the block named Who - values are");
      Put(CH09_1.Index, 5);                   -- CH09_1.Index
      Put(Index, 5);                          -- Who.Index
      Put(Who.Index, 5);                      -- Who.Index
      Put(Stuff, 5);                          -- Who.Stuff
      Put(Who.Stuff, 5);                      -- Who.Stuff
      Put(Count, 5);                          -- CH09_1.Count
      New_Line;
   end Who;

   Put("Back to the main block - values are");
   Put(Index, 5);                             -- CH09_1.Index
   Put(Count, 5);                             -- CH09_1.Count
   New_Line;

end CH09_1;




-- Result of execution

-- In the main block      - values are   27   33
-- In the embedded block  - values are   27  157 -345   33
-- This is in the new block procedure
-- Back to the main block - values are   27   33
-- In the block named Who - values are   27  157  157 -345 -345   33
-- Back to the main block - values are   27   33
```

2. Name the block in e_c09_p4.ada and output the value of the outer **Index** in the loop using the dot notation. It will output the same number six times since the outer **Index** is invariant in the inner loop.(Solution)

```
                        -- Chapter 9 - Programming exercise 2
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure CH09_2 is

   Dog, Cat : INTEGER;
   Pig, Cow : FLOAT;

begin
   My_Loop:
   for Index in 1..10 loop
      Put("The value of Index is");
      Put(Index, 3);

      declare
         START : constant INTEGER := Index;
         STOP  : constant INTEGER := START + 5;
         Count_Stuff : INTEGER;
      begin
         Count_Stuff := START + STOP + Index + 222;
         Put(" --->");
         for Index in START..STOP loop
            Put(My_Loop.Index, 5);
         end loop;
      end;

      New_Line;
   end loop My_Loop;

end CH09_2;




-- Result of execution

-- The value of Index is  1 --->    1    1    1    1    1    1
-- The value of Index is  2 --->    2    2    2    2    2    2
-- The value of Index is  3 --->    3    3    3    3    3    3
-- The value of Index is  4 --->    4    4    4    4    4    4
-- The value of Index is  5 --->    5    5    5    5    5    5
-- The value of Index is  6 --->    6    6    6    6    6    6
-- The value of Index is  7 --->    7    7    7    7    7    7
-- The value of Index is  8 --->    8    8    8    8    8    8
-- The value of Index is  9 --->    9    9    9    9    9    9
-- The value of Index is 10 --->   10   10   10   10   10   10
```

Ada Tutorial - Chapter 10

ARRAYS

OUR FIRST ARRAY

Example program ------> e_c10_p1.ada

```
                                       -- Chapter 10 - Program 1
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure Array1 is

   N : INTEGER := 10;
   Dummy1 : array(INTEGER range 1..7) of BOOLEAN;
   Dummy2 : array(INTEGER range -21..N) of BOOLEAN;
   Dummy3 : array(-21..N) of BOOLEAN;

   type MY_ARRAY is array(1..5) of INTEGER;

   Total        : MY_ARRAY;
   First        : MY_ARRAY;
   Second       : MY_ARRAY;
   Funny        : array(1..5) of INTEGER;
   X,Y          : array(12..27) of INTEGER;
   Fourth_Value : INTEGER renames First(4);

begin
   First(1) := 12;
   First(2) := 16;
   First(3) := First(2) - First(1);
   Fourth_Value := -13;
   First(5) := 16 - 2 * First(2);

   for Index in 1..5 loop
      Second(Index) := 3 * Index + 77;
   end loop;

   Total := First;
   if Total = First then
      Put("Both arrays are the same size and contain ");
      Put_Line("the same values in all elements.");
   end if;

   for Index in 1..5 loop
      Total(Index) := Total(Index) + Second(Index);
      Funny(Index) := Total(Index) + First(6 - Index);
      Put("The array values are");
      Put(Total(Index), 6);
      Put(Funny(Index), 6);
      New_Line;
   end loop;
end Array1;




-- Result of execution

-- Both arrays are the same size and contain the same values in...
-- The array values are    92    76
-- The array values are    99    86
-- The array values are    90    94
-- The array values are    76    92
-- The array values are    76    88
```

An array is a group of two or more elements that are all of the same type. In Ada, as in most modern computer languages, arrays can be made of many different kinds of data, but all elements of an array must be of the same type. The best way to see this is to inspect the program named e_c10_p1.ada which contains a few examples of arrays.

HOW DO WE DECLARE A SUBSCRIPT?

For simplicity, we will start with line 8 where we have a declaration of the array named **Dummy1**. This line says that the variable named **Dummy1** will have 7 elements numbered from 1 through 7, and each element will have the ability to store one **BOOLEAN** variable. We will see shortly that the individual elements will be referred to by the variable name followed by a subscript in parentheses, or **Dummy1(1)**, **Dummy1(2)**,... **to Dummy1(7)**. Keep in mind that each is a single **BOOLEAN** variable.

To define an array, we use the reserved words **array** and **of** with the appropriate modifiers as illustrated in this example. We define a range which the array will cover, the type of the range variable, which must be composed of discrete type limits, and the type of each element of the array. Remember that a discrete type is any type of the integer class including enumerated types. We will have an example program with an enumerated array index in part 2 of this tutorial.

LET'S LOOK AT ANOTHER ARRAY DECLARATION

In line 9, we have **Dummy2** defined as an array of **BOOLEAN** type variables that covers the range of **Dummy2(-21)** through **Dummy2(10)**, since **N** has the value of 10. Line 10 illustrates declaring the array **Dummy3** as an array of **BOOLEAN** variables from **Dummy3(-21)** through **Dummy3(10)**, but this time the subscript type is not explicitly stated, only implied by the values of the subscript limits. Actually, the subscript type was not needed in the first two either, since they were implied. The type **INTEGER** is used so often for an array subscript that the type **INTEGER** has been defined as the default if none is given. This is done only to make it a little simpler to use arrays.

So far in this program, we have defined about 70 variables that have no initial values because we have not assigned them any. We will see how to initialize an array later, but first we will learn how to use them.

HOW DO WE DEFINE AN ARRAY TYPE?

Line 12 gives the general method of declaring an array type. It uses the reserved word **type** followed by the type name, the reserved word **is**, then the definition of the type. The definition is composed of the range, followed by the reserved word **of** and the element type. We now have a type name which can be used to define any number of array variables, and each will be made up of integer variables. Each will also have 5 elements and will cover the range of 1 through 5. Thus line 14 defines an array of five elements named **Total**, the elements of which will be named **Total(1)**, **Total(2)**, ... **Total(5)**. Each of these elements can be used to store one value of type **INTEGER**.

Line 22 of the program illustrates how we can assign a value of 12 to one of the elements of **First**, which is another array of type **MY_ARRAY** and therefore composed of 5 elements. The second element is assigned the value of 16, and the third is assigned a value found by subtracting the first element from the second, resulting in a value of 4. This illustrates the use of the elements once they are assigned values. The fourth and fifth elements are also assigned nonsense data, illustrating some mathematical operations. The assignment in line 25 will be explained in the next paragraph. If you remember that each element is an **INTEGER** type variable, you can use them just like any other **INTEGER** type variable, except that you must add the subscript to indicate which element you are interested in using at each point in the program.

RENAMING AN ARRAY ELEMENT

Line 19 is an illustration of renaming a single element of the array. Once again, this simply gives us a synonym which we can use to refer to the variable, it does not declare another variable. It should be pointed out that it is not permitted to rename a type but you can achieve the same effect by declaring a subtype with the same range as the parent type.

WHAT IS A RANGE_ERROR?

Any attempt to use a subscript which is outside of the assigned range will result in the system raising the exception **Range_Error**, and will be handled as a fatal error, terminating operation of the program unless you handle the exception yourself. We will study exceptions in part 2 of this tutorial.

The subscript can be a variable itself, provided it is of the correct type, as is illustrated in line 29, where all five elements of the array named **Second** are assigned nonsense data for illustration. The subscript can also be calculated, with any arbitrary level of complexity, provided of course that it results in an **INTEGER** type result and is within the range of the declared subscript.

ARRAY ASSIGNMENT

The assignment in line 32 is legal, provided the two arrays are of the same type, resulting in all 5 values of the array named **First** being assigned to the five elements of the array named **Total**. Line 33 illustrates that arrays can even be compared for equality or inequality, and they are considered equal if each element of **Total** is equal to the corresponding element of **First**. If there is any inequality, the result is **FALSE**.

Lines 38 through 45 illustrate a few more of the permissible operations on arrays. You should have no trouble studying them on your own.

WHAT IS AN ANONYMOUS TYPE?

In line 12, we defined a type and gave it a name which we could then use at will throughout the remainder of the program. Any array of this type is said to be of type **MY_ARRAY**, because it has a name assigned to it. The array declared in line 17 does not have a name associated with it, so it is referred to as an anonymous type.

The array assignment in line 32 was only possible because the two arrays were of the same exact type, and in order to be of the same type, they must be declared with the same type name. The array named **Funny** has the identical structure as the array **First**, but it was not declared with the same type name, and it is therefore of a different type and cannot be used in an array assignment statement. Since the array **Funny** is of an anonymous type, it is impossible to define another array with the same type, so it is impossible to use this array in an assignment statement with any other array.

Line 18 declares the arrays **X** and **Y** in the same statement, and it would seem that they should have assignment compatibility, but according to the definition of Ada, the two arrays are of different types because naming both variable names in one statement is merely a shorthand method for naming them in two separate lines. The two arrays are therefore each of a separate anonymous type. This is a fine point, but should be clearly understood.

Two arrays are assignment compatible only if they are declared with the same type name. Compile and run this program and compare the output with the output you expect.

WHAT IS A SLICE OF AN ARRAY?

Example program ------> **e_c10_p2.ada**

```
                                       -- Chapter 10 - Program 2
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;
```

```
procedure Slice is

   type MY_ARRAY is array(1..12) of INTEGER;

   First, Second  : MY_ARRAY;
   Funny          : array(1..33) of INTEGER;

begin

   for Index in 1..33 loop
      Funny(Index) := 100 + Index * 2;
   end loop;

   for Index in 1..12 loop
      First(Index) := Index;
   end loop;

   Second(1..5) := First(3..7);
   Second(1..4) := First(6..9);
   Second(7..12) := First(3..8);
   First(2..9) := First(5..12);

   First(1..12) := MY_ARRAY(Funny(1..12));
   First(1..12) := MY_ARRAY(Funny(16..27));

   for Index in 1..12 loop
      Put("The array named First has the values ");
      Put(First(Index));
      New_Line;
   end loop;
end Slice;




-- Result of execution

-- The array named First has the values    132
-- The array named First has the values    134
-- The array named First has the values    136
-- The array named First has the values    138
-- The array named First has the values    140
-- The array named First has the values    142
-- The array named First has the values    144
-- The array named First has the values    146
-- The array named First has the values    148
-- The array named First has the values    150
-- The array named First has the values    152
-- The array named First has the values    154
```

The program named e_c10_p2.ada contains several examples of the use of the slice in Ada, which is a portion of an array. You may wish to assign part of an array to part of another array in a single statement. This can be done with a slice.

We begin by declaring an array type, **MY_ARRAY**, which is then used to declare two arrays, **First** and **Second**. Finally we declare a third array named **Funny**, which is of an anonymous type, which we explained during our study of the last example program. The current example program will illustrate the difficulty of working with an array of anonymous type, but we will start by working with the named arrays.

WHAT IS A SLICE?

In the executable part of the program, we assign nonsense values to the arrays named **Funny** and **First**, so we will have some data to work with. Then in line 22 we tell the system to take elements 3 through 7 of the array named **First** and assign them to elements 1 through 5 of the array named **Second**. The term on each side of the assignment operator is a slice, a portion of an array. In order to do the slice assignment as illustrated, both arrays must be of the same type and both slices must have the same number of elements. Of course, all slice limits must be within the declared range limits of the subscripts for the type in use. Line 23 illustrates copying 4 elements from **First** to **Second**, and line 24 illustrates copying 6 elements.

In line 25, eight elements are copied from an array to itself in such a manner that the destination and origin portions of the array overlap. Ada is defined such that all of the values are copied in true fashion rather than recopying some earlier copied values again as the copying of values continues. This is because the entire right hand expression is evaluated before the assignment is made to the left side variable. Note that the slice can only be used with a singly dimensioned array.

BACK TO THE ANONYMOUS TYPE VARIABLE

We said that the variable named **Funny** is an anonymous type and that it would cause some difficulties, so let's see what the problems are. In order to assign all or part of an array to another type of array, we must use a type transformation or get a compile error. Line 27 illustrates how to use a type transformation, as we have seen before. However, we can only do a type transformation based on the entire array, not a portion of it, so we can only transform the type from anonymous to a full sized array of the target type, **MY_ARRAY**. Therefore we can only copy a slice from the anonymous type variable into the full array of the target. There is no way to transform the type from **MY_ARRAY** to the anonymous type, since the anonymous type doesn't have a name, so a slice cannot be used to assign to the array variable named **Funny**. Line 28 illustrates another slice assignment to the complete array named **First**.

The entire array named **First** is displayed for your observation. Compile and run this program and examine the output.

A MULTI DIMENSIONAL ARRAY

Example program ------> **e_c10_p3.ada**

```
                                       -- Chapter 10 - Program 3
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure MultAry1 is

   type MATRIX is array(INTEGER range 1..3,
                        INTEGER range 1..4) of INTEGER;

   Square_Board  : MATRIX;
   Checker_Board : MATRIX;
   Chess_Board   : array(INTEGER range 1..3,
                         INTEGER range 1..4) of INTEGER;
   Across, Over  : INTEGER;

begin

   for Across in 1..3 loop
      for Over in 1..4 loop
         Square_Board(Across, Over) := Across * Over;
         Chess_Board(Across, Over) := 0;
      end loop;
   end loop;
   Checker_Board := Square_Board;

   Checker_Board(2, 3) := 2;
   Checker_Board(Checker_Board(2, 3), 4) := 17;
   Checker_Board(3, 3) := Chess_Board(3, 3);

   for Across in 1..3 loop
      for Over in 1..4 loop
         Put(Checker_Board(Across, Over), 5);
      end loop;
      New_Line;
   end loop;

end MultAry1;




-- Result of execution

--      1    2    3    4
--      2    4    2   17
--      3    6    0   12
```

Examine the program named e_c10_p3.ada for our first example of a multidimensional array. We begin by declaring a type named **MATRIX** which is composed of an array of an array with a total of 12 elements. Each element of the array is referred to by two subscripts following the variable name in parentheses as illustrated in the executable part of the program. Lines 18 through 23 contain a nested loop to fill the variable named **Square_Board** with a multiplication table, and to fill **Chess_Board** with all zeros. Note that the variable named **Chess_Board** is of an anonymous type because there is no type name associated with it.

The entire array named **Square_Board** is assigned to the array **Checker_Board** in line 25, which is legal because they are of the same type, which means they were defined with the same type name. Line 27 is used to assign a value of 2 to one element of the **Checker_Board**, and that value is used in line 28, which states, "Checker_Board(2,4) := 17;", because line 27 assigned the value of 2 to **Checker_Board(2,3)**.

It should be clear to you, based on the discussion of the last program, that even though **Chess_Board** has the same structure as **Square_Board**, they are not type compatible and are not assignment compatible. The individual elements are assignment compatible however.

The array named **Checker_Board** is displayed which you can observe when you compile and run this program. By the way, the array names are poorly chosen in this example because a chess board is not 3 by 4. Good naming conventions help toward developing quality software.

WE NEED SOME FLEXIBILITY

Example program ------> **e_c10_p4.ada**

```
                                       -- Chapter 10 - Program 4
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure MultAry2 is

   SIZE : constant := 3;
   NEXT : constant := SIZE + 1;

   type MATRIX is array(INTEGER range 1..SIZE,
                          INTEGER range 1..NEXT) of INTEGER;

   Square_Board  : MATRIX;
   Checker_Board : MATRIX;
   Chess_Board   : array(INTEGER range 1..SIZE,
                          INTEGER range 1..NEXT) of INTEGER;
   Across, Over  : INTEGER;

begin

   for Across in 1..SIZE loop
      for Over in 1..NEXT loop
         Square_Board(Across, Over) := Across * Over;
         Chess_Board(Across, Over) := 0;
      end loop;
   end loop;

   Checker_Board := Square_Board;

   Checker_Board(2, 3) := 2;
   Checker_Board(Checker_Board(2, 3), 4) := 17;
   Checker_Board(3, 3) := Chess_Board(3, 3);

   for Across in 1..SIZE loop
      for Over in 1..NEXT loop
         Put(Checker_Board(Across, Over), 5);
      end loop;
      New_Line;
   end loop;

end MultAry2;




-- Result of execution

--      1    2    3    4
--      2    4    2   17
--      3    6    0   12
```

The previous program used fixed values for all of the range and loop definitions, which allows very little flexibility, and is therefore considered to be poor programming practice. Of course, it was done for clarity since this was your first look at a multidimensional array. The next program, named e_c10_p4.ada is much more flexible and illustrates some of the slightly more advanced techniques which can be used with Ada.

This program is identical to the last except that there are two constants defined in the declaration part which are then used to define the limits of the arrays and the loops. If you needed to make the program cover a larger range, it would be trivial to modify the constants and recompile the program. Compile and run this program and you will see that it does exactly the same thing as the last one.

WE NEED MORE FLEXIBILITY

Example program ------> **e_c10_p5.ada**

```
                                       -- Chapter 10 - Program 5
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure MultAry3 is
```

```
   SIZE : constant := 3;
   NEXT : constant := SIZE + 1;

   type MATRIX is array(INTEGER range 1..SIZE,
                        INTEGER range 1..NEXT) of INTEGER;

   Square_Board  : MATRIX;
   Checker_Board : MATRIX;
   Chess_Board   : array(INTEGER range 1..SIZE,
                         INTEGER range 1..NEXT) of INTEGER;
   Across, Over  : INTEGER;

begin

   for Across in 1..Square_Board'LAST(1) loop
      for Over in 1..Square_Board'LAST(2) loop
         Square_Board(Across, Over) := Across * Over;
         Chess_Board(Across, Over) := 0;
      end loop;
   end loop;

   Checker_Board := Square_Board;

   Checker_Board(2, 3) := 2;
   Checker_Board(Checker_Board(2, 3), 4) := 17;
   Checker_Board(3, 3) := Chess_Board(3, 3);

   for Across in Checker_Board'RANGE(1) loop
      for Over in Checker_Board'RANGE(2) loop
         Put(Checker_Board(Across, Over), 6);
      end loop;
      New_Line;
   end loop;

end MultAry3;




-- Result of execution

--      1     2     3     4
--      2     4     2    17
--      3     6     0    12
```

Examine the program named e_c10_p5.ada and you will find that it is identical to the last two except in the way we define the loop limits. Recall the information we covered on attributes in an earlier lesson and the additions to this program will be simple for you to understand. In line 21 we use the attribute **LAST** to define the upper limit of the outer loop and add the digit "1" in parentheses to tell the system that we are interested in the first subscript of the array named **Square_Board**. Line 22 uses a "2" to indicate the **LAST** value of the second subscript of **Square_Board**. If no subscript indication is given, the system will default to "1", but it is much clearer for the reader to indicate the "1" in parentheses. It may seem to you to be a lot of trouble to define the limits this way, but when we get to the point where we are writing generalized procedures we will need the flexibility given here. Generic procedures are a long way off too, but these techniques will be absolutely essential when we study them.

Lines 34 and 35 also use attributes for the two loop ranges, but they use the **RANGE** attribute with the number of the desired subscript in parentheses once again.

When using a singly subscripted array, it is legal to use a "1" in parentheses also, but if none is given, the system will default to one. You should explicitly include the number in a multidimensional array and omit it for a singly dimensioned array as a matter of programming clarity.

HOW DO WE INITIALIZE ARRAYS?

Example program ------> **e_c10_p6.ada**

```
                                       -- Chapter 10 - Program 6
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure ArryInit is

   type MY_ARRAY is array(1..5) of INTEGER;

   Total : MY_ARRAY := (12, 27, -13, 122, 44);
   First : MY_ARRAY := (1 => 14, 2 => 57, 3=> 111,
                              5 => -27, 4 => 21);

   Another : MY_ARRAY := (2 => 13, 5 => 57, others => 3);

   One_More : MY_ARRAY := (2..4 => 13, 1 | 5 => 27);

   type MATRIX is array(INTEGER range 1..3,
                        INTEGER range 1..4) of INTEGER;

   Square_Board : MATRIX := ((4, 7, 3, 5),
                             (3, 8, 2, 0),
                             (1, 5, 9, 9));

   Checker_Board : MATRIX := (2 => (3, 8, 2, 0),
                              3 => (1, 5, 9, 9),
                              1 => (4, 7, 3, 5));

   Chess_Board : MATRIX := (2 => (3, 8, 2, 0),
                            3 => (1, 5, 9, 9),
                            1 => (4 => 5, 2 => 7, 1 => 4, 3 => 3));

begin

   if Square_Board = Checker_Board then
      Put_Line("The two arrays are equal.");
   end if;

   if Chess_Board = Square_Board then
      Put_Line(" and so are the other two.");
   end if;

end ArryInit;




-- Result of execution

-- The two arrays are equal.
--  and so are the other two.
```

Examine the program named e_c10_p6.ada for some examples of array initialization. Seven arrays are declared and the method of aggregate notation, both positional and named, are illustrated. An aggregate is a group of numeric literals, although enumeration values could also be included, that are used in many places in Ada. We will use an aggregate to initialize an array in this example program. The literals can be grouped in the order in which they are used, and this is referred to as a positional aggregate. The literals can also be in a named aggregate, in which the use for each value is defined by using the name of the location to which it should be assigned.

The variable **Total** is initialized using the positional notation and the variable **First** is initialized by use of the named notation. Mixed notation is not allowed for array initialization. The array named **Another** in line 13 contains a new construct using the reserved word **others** in conjunction with the named aggregate notation. If included, it must be the last entry in the aggregate. The values of **Another(1)**, **Another(3)**, and **Another(4)** will be initialized to the value of 3. The array named **One_More** in line 15 illustrates initialization of a range of variables to the value 13, and two variables to the value 27. Note that the **others** case can be included here but must be last and alone. The other singly dimensioned arrays should pose no problem for you but a few comments are in order concerning the multi dimensional arrays.

Even though you are not permitted to use mixed aggregate notation for an array, the rule is applied at only one level so you can use different methods for each level.

The variable **Square_Board** uses all positional notation, but **Checker_Board** uses a named aggregate for the first subscript and positional for the second. **Chess_Board** mixes things up a little more by using a named aggregate for the first subscript and both methods for the second subscript even though it is consistent within each subgroup and is therefore obeying the rules.

MORE ARRAY EXAMPLES LATER

There is more to be said about arrays, but it will have to wait until we cover a few more topics. This is meant to get you started using arrays, but in a later chapter we will cover additional array topics.

PROGRAMMING EXERCISES

1. Modify e_c10_p1.ada in such a way that the variables **X** and **Y** are assignment compatible. (Solution)

```
                                   -- Chapter 10 - Programming exercise 1
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure CH10_1 is

   N : INTEGER := 10;
   Dummy1 : array(INTEGER range 1..7) of BOOLEAN;
   Dummy2 : array(INTEGER range -21..N) of BOOLEAN;
   Dummy3 : array(-21..N) of BOOLEAN;

   type MY_ARRAY is array(1..5) of INTEGER;
   type NEW_ARRAY_TYPE is array(12..27) of INTEGER;

   Total        : MY_ARRAY;
   First        : MY_ARRAY;
   Second       : MY_ARRAY;
   Funny        : array(1..5) of INTEGER;
   X,Y          : NEW_ARRAY_TYPE;
   Fourth_Value : INTEGER renames First(4);

begin
   for Index in 12..27 loop
      X(Index) := Index + 7;
   end loop;
   Y := X;           -- These are now assignment compatible

   First(1) := 12;
   First(2) := 16;
   First(3) := First(2) - First(1);
   Fourth_Value := -13;
   First(5) := 16 - 2*First(2);

   for Index in 1..5 loop
      Second(Index) := 3 * Index + 77;
   end loop;

   Total := First;
   if Total = First then
      Put("Both arrays are the same size and contain ");
      Put_Line("the same values in all elements.");
   end if;

   for Index in 1..5 loop
      Total(Index) := Total(Index) + Second(Index);
      Funny(Index) := Total(Index) + First(6 - Index);
      Put("The array values are");
      Put(Total(Index), 6);
      Put(Funny(Index), 6);
      New_Line;
   end loop;
end CH10_1;




-- Result of execution

-- Both arrays are the same size and contain the same values in...
-- The array values are    92    76
-- The array values are    99    86
-- The array values are    90    94
-- The array values are    76    92
-- The array values are    76    88
```

2. Write a program with two arrays of size 3 by 5 each and initialize each to a suitable set of integer values. Multiply these, element by element, and store the values in a third array. Finally, display the results in a clear format on the monitor.(Solution)

```
                           -- Chapter 10 - Programming exercise 2
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure Ch10_2 is

type MATRIX is array(1..3,1..5) of INTEGER;

First  : MATRIX := ((1, 1, 1, 1, 1),
                    (2, 2, 2, 2, 2),
                    (3, 3, 3, 3, 3));

Second : MATRIX := ((1, 2, 3, 4, 5),
                    (1, 2, 3, 4, 5),
                    (1, 2, 3, 4, 5));
```

```
Result : MATRIX;

begin
   for Index1 in 1..3 loop
      for Index2 in 1..5 loop
         Result(Index1, Index2) := First(Index1, Index2) *
                                   Second(Index1, Index2);
      end loop;
   end loop;

   for Index1 in 1..3 loop
      for Index2 in 1..5 loop
         Put(Result(Index1, Index2), 4);
      end loop;
      New_Line;
   end loop;
end Ch10_2;




-- Result of execution

--     1   2   3   4   5
--     2   4   6   8  10
--     3   6   9  12  15
```

Ada Tutorial - Chapter 11

THE CHARACTER AND STRING TYPE

A QUICK REVIEW OF THE CHARACTER TYPE

Example program ------> **e_c11_p1.ada**

```
                                       -- Chapter 11 - Program 1
with Ada.Text_IO;
use Ada.Text_IO;

procedure Chars is

   My_Char : CHARACTER;
   Another : CHARACTER := 'D';

begin

   My_Char := 'A';
   if My_Char < Another then
      Put_Line("My_Char is less than Another.");
   end if;

   Put(My_Char);
   Put(Another);
   Put(My_Char);
   Put_Line(" character output.");

   My_Char := CHARACTER'SUCC(My_Char);            -- 'B'
   My_Char := CHARACTER'FIRST;                    -- nul code
   My_Char := CHARACTER'LAST;                     -- del code

end Chars;




-- Result of execution

-- My_Char is less than Another.
-- ADA character output.
```

The best way to study any topic is with an example, so examine the program named e_c11_p1.ada for some examples using **CHARACTER** type variables.

The type **CHARACTER** is a predefined type in Ada and is defined as the printable set of ASCII characters including a few that don't actually print. See Annex A.1 of the Ada 95 Reference Manual (ARM) for a complete list of the **CHARACTER** elements. All of the operations available with the enumerated type variable are available with the **CHARACTER** type variable. To illustrate their use, we declare two **CHARACTER** type variables in lines 7 and 8 with the second being initialized to the letter D. Note the single quote marks which define the **CHARACTER** type literal to which the variable named **Another** is initialized. A different literal value is assigned to the variable **My_Char** in line 12, and the two variables are compared in the **if** statement. Since 'A' is of lesser value than 'D', the line of text in line 14 is output to the monitor.

Lines 17 through 20 display some very predictable output that is included as an example of **CHARACTER** output, and finally some of the attributes available with the **CHARACTER** type variable are illustrated in lines 22 through 24. The same attributes are defined for the **CHARACTER** type variable as for the enumerated type and all are listed in Annex K of the ARM.

Compile and execute this program to get a feel for use of the **CHARACTER** type variable.

You may wish to review the program named e_c07_p3.ada in chapter 7 to refresh your mind on declaring subtypes and derived types of the predefined **CHARACTER** type.

THE STRING TYPE

Example program ------> **e_c11_p2.ada**

```
                                       -- Chapter 11 - Program 2
with Ada.Text_IO;
use Ada.Text_IO;

procedure String1 is

   Line     : STRING(1..33);
   NAME     : constant STRING := ('J','o','h','n');
   JOB      : constant STRING := "Computer Programmer";
   Address  : STRING(1..13) := "Anywhere, USA";
   Letter   : CHARACTER;
   EXAMPLE1 : constant STRING := "A";      -- A string of length 1
   EXAMPLE2 : constant STRING := "";       -- An empty string

begin

   Line := "This is a test of STRINGS in Ada.";
   Put(Line);
   New_Line;
   Put(NAME);
   Put(" is a ");
   Put(JOB);
   Put(" and lives in ");
   Put(Address);
   New_Line(2);

   Address(3) := 'X';                -- Individual letters
   Address(4) := 'Y';                -- Individual letters
   Address(10..13) := NAME(1..4);    -- A slice
   Put(Address);
   New_Line;

end String1;




-- Result of execution

-- This is a test of STRINGS in Ada.
-- John is a Computer Programmer and lives in Anywhere, USA
--
-- AnXYhere,John
```

The program named e_c11_p2.ada illustrates some of the operations that can be done with the predefined type **STRING**. A string is an array of **CHARACTER** type variables which is of a fixed length and starts with element number 1 or higher. The index uses type **POSITIVE**. Note that this program is called e_c11_p2.ada instead of the more desirable name of STRING.ADA because the word **STRING** is a predefined word in Ada and using it for the program name would make it unavailable for its correct use.

Line 7 declares an uninitialized string of 33 characters, while line 8 declares a constant string of four elements initialized to the word "John", and illustrates rather graphically that the string is composed of individual **CHARACTER** type elements. Line 9 declares another constant that is initialized, which all constants must be in order to be useful. Note that lines 8 and 9 did not contain a character count, the computer counted the characters for us and supplied the limits of the array.

DECLARING A STRING VARIABLE

Line 10 defines a **STRING** variable, which will be initialized to the string given. Even though the initialization string is given, the array limits must be explicitly specified for a variable. Not only must the limits be given, the number of elements in the initialization string must agree with the number of elements defined as the array range, or a compiler error will be given. This is the first difficulty encountered when using strings, but there will be more as we progress. It seems like the computer should be able to count the characters in the variable for us, but due to the strong type checking used in Ada, this cannot be done.

STRING MANIPULATION IS DIFFICULT

When we get to the executable part of the program, we assign a string constant to the string variable named **Line**. Once again, according to the definition of Ada, the string constant must have exactly the same number of characters as the number of characters in the declaration of the variable **Line**, or a compile error will be issued. This is another seemingly unnecessary inconvenience in the use of strings which we must put up with. The variable named **Line** is displayed on the monitor in line 18, and some of the other constants are displayed along with it. Note that the string literal in line 21 is simply another string constant, but it does not have a name. Finally, we assign data to a few individual elements of the string variable named **Address** in such a way to illustrate that it is indeed an array, then do a slice assignment, and finally output the result. It should be noted that the **Put_Line** could be used instead of the two separate output procedure calls in lines 30 and 31, but it is simply a matter of personal taste.

Compile and run this program and see that the output is exactly what you predict from your understanding of the program.

CONCATENATION OF STRINGS

Example program ------> **e_c11_p3.ada**

```
                                       -- Chapter 11 - Program 3
with Ada.Text_IO, Ada.Characters.Latin_1;
use Ada.Text_IO;

procedure Concat is

   String4 : STRING(1..4);
   String7 : STRING(1..7);

begin

   String7 := "CAT" & "FISH";                          -- CATFISH
   Put(String7); New_Line;

   String4 := "CAT" & "S";                             -- CATS
   Put(String4); New_Line;

   String4 := "S" & "CAT";                             -- SCAT
   Put(String4); New_Line;

   String7 := String4 & "cat";                         -- SCATcat
   Put(String7); New_Line;

   String7 := "Go" & Ada.Characters.Latin_1.CR &
                 Ada.Characters.Latin_1.LF & "Car"; -- Go
   Put(String7); New_Line;                           -- Car

   String7(3..5) := "ldb";                           -- Goldbar
   Put(String7); New_Line;

end Concat;




-- Result of execution

-- CATFISH
-- CATS
-- SCAT
-- SCATcat
-- Go
-- Car
-- Goldbar
```

Examine the program e_c11_p3.ada for several examples of string concatenation. Two uninitialized string variables are declared in lines 7 and 8, and they are used throughout the program.

Line 12 illustrates concatenation of a three element string and a four element string by using the concatenation operator, the "&". The four element string is appended to the end of the three element string forming a seven element string which is assigned to the variable **String7**. Line 21 illustrates concatenation of a four element variable with a three element constant.

Line 24 is the most interesting assignment here, because it is a concatenation of four strings, two of which contain only one element each. The values of "CR" and "LF" are such that they produce a "carriage return" and "line feed" when sent to the monitor, so that when **String7** is output, it will be on two successive lines of the monitor. The ASCII values of all of the characters are available in the predefined package named **Ada.Characters.Latin_1**, which is why the dotted notation gives the actual value of these constants. Use of the dot notation in this manner will be more fully defined later in this tutorial. Be sure to compile and run this program, and be sure you understand the results.

STRING COMPARISONS

Example program ------> **e_c11_p4.ada**

```
                                        -- Chapter 11 - Program 4
with Ada.Text_IO;
use Ada.Text_IO;

procedure StrngCmp is

   MY_CAR   : constant STRING := "Ford";
   YOUR_CAR : constant STRING := "FORD";
   Her_Car  : STRING(1..8) := "Mercedes";
   Rental   : STRING(1..4);
   Lease    : STRING(1..4);

begin

   if MY_CAR /= YOUR_CAR then
      Put_Line("Case matters in a STRING constant or variable");
   end if;

   Her_Car := "Ford    ";    -- This is still not equal to My_Car
   Rental := MY_CAR;
   Rental := "Junk";
   Lease := Rental;

   If Rental = "Junk" then
      Put_Line("A variable can be compared to a string literal.");
   end if;

end StrngCmp;




-- Result of execution

-- Case matters in a STRING constant or variable
-- A variable can be compared to a string literal.
```

The example program named e_c11_p4.ada will give you some examples of string comparisons as used in Ada, so you should examine it at this time. The string declarations are nothing new to you, so nothing more will be said about them.

In line 15 where the constants **MY_CAR** and **YOUR_CAR** are compared for inequality, they will not be equal since the case is different for some of the characters, and case matters in a string expression. A different ASCII value is used for the letter 'A' than that used for the letter 'a', so they are not the same. For a string comparison to be equal, all elements must be exactly the same as the corresponding elements in the other string, and the number of elements must be the same. Therefore, following execution of line 19, the value assigned to **Her_Car** is still not the same as the value stored in the constant **MY_CAR.** If you attempted to compare them, you would get a compile error because the two strings have a different length, so they could never compare anyway. Line 24 illustrates that a variable can be compared to a string literal.

Lines 20 through 22 are examples of legal statements according to the Ada definition. Compile and run this program and study the resulting output.

ATTRIBUTES OF CHARACTERS AND STRINGS

Example program ------> e_c11_p5.ada

```
                                       -- Chapter 11 - Program 5
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure CharInt is

   Char : CHARACTER;
   Index : INTEGER;
   Stuff : array(0..25) of CHARACTER;

begin
   Char := 'A';
   Index := 5 + CHARACTER'POS(Char);
   Put(Index, 5);
   Char := CHARACTER'VAL(Index);
   Put(Char);

   New_Line;
   Stuff(21) := 'X';
   Index := 2 + CHARACTER'POS(Stuff(21));
   Put(Index, 5);
   Stuff(0) := CHARACTER'VAL(Index);
   Put(Stuff(0));

end CharInt;




-- Result of execution

--    70F
--    90Z
```

Examine the program named e_c11_p5.ada for examples of how you can convert from **CHARACTER** type variables to **INTEGER** type variables and back. The attributes **POS** and **VAL** are used as shown. In order to increment a character, for example an 'A', to the next value, it must be converted to **INTEGER**, incremented, then converted back to **CHARACTER**. Of course you could always use the **SUCC** attribute to increment the **CHARACTER** type variable.

This program should be self explanatory. After you study it, compile and run it.

THERE ARE TWO KINDS OF STRINGS NOW

With the upgrade to Ada 95, there are now two kinds of strings. The **STRING** type that we have been discussing in this chapter, and a new **WIDE_STRING** type. Since there are far more than 256 different characters in some languages around the world, and Ada is approved by the International Standards Organization (ISO), it was necessary to provide the ability to handle many more characters. The **WIDE_CHARACTER** type was defined which provides 65,536 different characters, and the **WIDE_STRING** library was provided which uses the larger character type for each of its characters. The first 256 characters of the **WIDE_CHARACTER** type are the same as the characters in the **CHARACTER** type. The remaining characters can be defined as needed for whatever language is being used in any given application.

A NEW, VERY USEFUL LIBRARY PACKAGE

Ada 95 has a new character handling library defined for use in text based processing. The library named **Ada.Characters.Handling** is composed of many useful subprograms for use with text handling. It contains, for example, a function named **Is_Upper** which returns a **BOOLEAN** value indicating whether the character passed in as a parameter is upper case or not. Another function, **To_Upper** changes the case of the character passed in to upper case, if it is an alphabetic character. There are functions to check for whitespace, if a character is numeric, if it is a special character, and many other useful functions. The student is encouraged to study this package provided by your compiler, especially if text processing will be a major part of your programming efforts.

DYNAMIC STRINGS ARE COMING

You may not feel too good about the use of strings in Ada because of the lack of flexibility, but don't worry about them. Ada was written to be an extendable language and when we get to chapter 16, we will have an example package that will give you the ability to use strings the way you would like to. A rather extensive dynamic string package will be presented to you and you will have the ability to refine it even further if you so desire. In effect, you will have the ability to extend the Ada language.

Ada 95 has an improvement that was not available with Ada 83, the predefined string packages which will be covered later in this tutorial.

PROGRAMMING EXERCISES

1. Write a program to declare your first name as a **STRING** constant, and your last name as another **STRING** constant. Concatenate your first and last names with a blank between them

and display the result with a single **Put_Line**. You will find much inflexibility in the definition of the **STRING** variable you use for the result.(Solution)

```
                                    -- Chapter 11 - Programming exercise 1
with Ada.Text_IO;
use Ada.Text_IO;

procedure Ch11_1 is

First_Name : constant STRING := "John";
Last_Name  : constant STRING := "Doe";
Full_Name  : STRING(1..8);

begin

   Full_Name := First_Name & ' ' & Last_Name;
   Put_Line(Full_Name);

end Ch11_1;




-- Result of execution

-- John Doe
```

2. Add code to e_c11_p5.ada to increment the variable named **Char** by using the **POS** and **VAL** attributes. Then add additional code to increment the same variable through use of the **SUCC** attribute.(Solution)

```
                   -- Chapter 11 - Programming exercise 2
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure CH11_2 is

   Char : CHARACTER;
   Index : INTEGER;
   Stuff : array(0..25) of CHARACTER;

begin
   Char := 'A';
   Index := 5 + CHARACTER'POS(Char);
   Put(Index, 7);
   Char := CHARACTER'VAL(Index);
   Put(Char);

   New_Line;
   Stuff(21) := 'X';
   Index := 2 + CHARACTER'POS(Stuff(21));
   Put(Index, 7);
   Stuff(0) := CHARACTER'VAL(Index);
   Put(Stuff(0));

   New_Line;
   Char := CHARACTER'VAL(CHARACTER'POS(Char) + 1);
   Put(Char);
   Char := CHARACTER'SUCC(Char);
   Put(Char);
```

```
end CH11_2;



-- Result of execution

--      70F
--      90Z
-- GH
```

Ada Tutorial - Chapter 12

RECORDS

OUR FIRST LOOK AT A RECORD

Example program ------> e_c12_p1.ada

```
                                       -- Chapter 12 - Program 1
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure Record1 is

   type DATE is
      record
         Month : INTEGER range 1..12;
         Day   : INTEGER range 1..31;
         Year  : INTEGER range 1776..2010;
      end record;

   Independence_Day : DATE;
   Birth_Day        : DATE;
   Today,Pay_Day    : DATE := (5, 25, 1982);

begin
   Independence_Day.Month := 7;
   Independence_Day.Day := 4;
   Independence_Day.Year := 1776;

   Birth_Day := Independence_Day;

   Pay_Day.Day := 30;

   Put("Independence day was on ");
   Put(Independence_Day.Month, 2);
   Put("/");
   Put(Independence_Day.Day, 2);
   Put("/");
   Put(Independence_Day.Year, 4);
   New_Line;

   Birth_Day := (Day => 19, Month => 2, Year => 1937);
   Today := (7, 14, 1952);
   Pay_Day := (7, Year => 1954, Day => 17);

end Record1;




-- Result of execution

-- Independence day was on  7/ 4/1776
```

Ada has provision for two composite types, the array, which we studied earlier, and the record, which is the topic of this chapter. Examine the program named e_c12_p1.ada for our first example of a record.

Lines 7 through 12 declare an Ada record, which actually only declares a type. The usual syntax for a type is used but when we come to the type definition itself, we begin with the reserved word

record. The components of the record are then inserted, and the record type definition is terminated by the reserved words **end record.**

A record can contain any desired components, the only requirement being that the component types must be declared prior to this definition, and the record type cannot be included as a component of itself. The key point to keep in mind about records is, whereas an array is composed of some number of like elements, the record is composed of some number of components that may be of different types.

WHAT IS CONTAINED IN THE RECORD?

It is impossible to declare an anonymous record type like you can do in Pascal. The record must be a named type prior to being used in a variable declaration.

In this record, we have a variable named **Month** that is permitted to store any value from 1 through 12, obviously representing the months of the year. There are also **Day** and **Year** variables, each of which is different from **Month** since different constraints are placed upon each. After declaring the record type, we still have no actual variables, only a type, but in lines 14 through 16, we declare four variables of type **DATE**. Since the variables are of type **DATE**, each has three components, namely a **Month**, **Day**, and **Year**. Notice that two of the variables are initialized to the values given in parentheses in the order of the variable definitions. **Month** is therefore set to 5, **Day** to 25, and **Year** to 1982 for each of the two initialized variables, **Today** and **Pay_Day**. The initialization will be very clear after we discuss the program itself, so we will come back to it later.

HOW DO WE USE THE RECORDS?

Since **Independence_Day** is actually a variable composed of three different variables, we need a way to tell the computer which subfield we are interested in using. We do this by combining the major variable name and the subfield with a dot as shown in lines 19 through 21. This is called the selected component notation in Ada. It should be clear to you that **Independence_Day.Month** is actually a single variable capable of storing an **INTEGER** type number as long as it is in the range of 1 through 12. The three elements of the record are three simple variables that can be used in a program wherever it is possible to use any other integer type variable. The three are grouped together for our convenience because they define a date which we call **Independence_Day**. The data could be stored in three simple variables and we could keep track of them in the way we usually handle data, but the record allows a more convenient grouping and a few additional operations.

THE RECORD ASSIGNMENT

There is one big advantage to using a record and it is illustrated in line 23 where all three values associated with the variable **Independence_Day** are assigned to the three corresponding components of the variable **Birth_Day**. If they were separate variables, they would have to be copied one at a time. The **Day** field of **Pay_Day** is assigned a new value in line 25 and the date contained in **Independence_Day** is displayed on the monitor for illustrative purposes.

NAMED AND POSITIONAL AGGREGATES

Line 35 has an example of assignment using a named aggregate in which the three fields are defined with their respective names and the pointing operator. It can be read as, "the variable named **Day** gets the value of 19, **Month** gets the value of 2, and so on". Since they are named, they are not required to be in the same order that they are in the record definition, but can be in any order. The real advantage to using the named aggregate notation is the fact that all elements are named and it is clear just what value is being assigned to each variable field. It should be pointed out that an aggregate is a group of data which may or may not be of the same type.

Line 36 defines the three values of the record by simply giving the three values, but in this case, the three elements must be in the correct order so the compiler will be able to assign them to their correct subfields. This is called a positional aggregate. This is the kind of aggregate used to initialize the dates in line 16.

A MIXED AGGREGATE

Line 37 illustrates use of a mixed aggregate in which some are defined by their position, and the rest are defined by their names. The positional definitions must come first, and after a named variable is given, the remainder must be named also. One point that must be remembered, all values must be mentioned, even if some of them will not be changed. This seems like a picky nuisance, but it greatly simplifies the compiler writer's job.

Compile and run this program, and you will get the date of Independence Day displayed on your monitor. Be sure you understand this program, because understanding the next program requires that you thoroughly understand this one. It should be clear that whether you use the named, positional, or mixed notation, you are required to use the correct types for each of the parameters.

A RECORD CONTAINING A RECORD

Example program ------> e_c12_p2.ada

```
                                       -- Chapter 12 - Program 2
with Ada.Text_IO;
use Ada.Text_IO;

procedure Record2 is

   type DATE is
      record
         Month : INTEGER range 1..12;
         Day   : INTEGER range 1..31;
         Year  : INTEGER range 1776..2010;
      end record;

   type PERSON is
      record
         Name      : STRING(1..15);
         Birth_Day : DATE;
         Age       : INTEGER;
         Sex       : CHARACTER;
      end record;

   Self, Mother, Father : PERSON;

   My_Birth_Year : INTEGER renames Self.Birth_Day.Year;

begin

   Self.Name := "John Q. Doe    ";
   Self.Age := 21;
   Self.Sex := 'M';
   Self.Birth_Day.Month := 10;
   Self.Birth_Day.Day := 18;
   Self.Birth_Day.Year := 1938;
   My_Birth_Year := 1938;        -- Identical to previous statement

   Mother := Self;
   Father.Birth_Day := Mother.Birth_Day;
   Father.Birth_Day.Month := Self.Birth_Day.Month - 4;
   Mother.Sex := 'F';

   if Mother /= Self then
      Put_Line("Mother is not equal to Self.");
   end if;
end Record2;




-- Result of execution

-- Mother is not equal to Self.
```

Examine the file named e_c12_p2.ada for an example of a record declaration containing another record within it. We declare the record type **DATE** in exactly the same manner that we did in the last program, but we go on to declare another record type named **PERSON**. You will note that the new record is composed of four variables, one of the variables being of type **DATE** which contains three variables itself. We have thus declared a record that contains three simple variables and a variable record containing three more variables, leading to a total of six separate variables within this one record type. In line 22, we declare three variables, each composed of six simple variables, so we have 18 declared variables to work with in our example program.

HOW TO USE THE COMPOSITE RECORD

Lines 28 through 30 should pose no real problem for you since we are using knowledge gained during the last example program, but to assign the date requires another extension to our store of Ada knowledge. Notice that in addition to the name of the main variable **Self**, we must mention the **Birth_Day** variable which is part of it, and finally the subfield of the **Birth_Day** variable, **Month** in line 31. The variable name is therefore composed of the three names, "dotted" together resulting in the name of a unique simple variable. Once again, this is called the selected component notation. Lines 32 through 34 assign the remaining three fields of the variable **Self** some meaningful data. Line 36 assigns all six elements of the variable **Self** to the variable **Mother** in one simple statement. Line 37 is used to assign the values of only the three components of **Mother**'s **Birth_Day** to the three corresponding components of **Father**'s **Birth_Day**.

Since each subfield is actually a simple variable, each one can be used in computations as illustrated in line 38 where **Mother**'s **Birth_Day Month** is assigned the value which is 4 less than **Self**'s **Birth_Day Month**. This is only done to illustrate that the simple variables can be used in any way you so desire, provided that you follow the rules of simple types.

RENAMING A RECORD COMPONENT

Line 24 illustrates how you can rename a component of a record in order to ease the problem of entering the dotted notation each time a field is used. In this case the simple name **My_Birth_Year** is a synonym for the extended naming required with all three components. Once again it must be pointed out that this only affects speed of compilation since it is only an additional name which can be used to refer to the variable. It must also be repeated that this facility should not be used except in those few case where it really adds to the program clarity. Correct use of the new name is illustrated in line 34.

RECORD ASSIGNMENT AND COMPARISON

As illustrated in lines 36, 37, and 41, entire records can be assigned to other records of the same type, and entire records of the same type can be compared for equality or inequality. The records are equal only if every subfield of one record is equal to the corresponding subfield of the other. The other comparison operators are not available in Ada for records.

Compile and run this program even though you will not get any output. Add some output statements yourself to see if you can get some of the data out to the monitor.

AN ARRAY WITHIN A RECORD

Example program ------> **e_c12_p3.ada**

```
                                       -- Chapter 12 - Program 3
with Ada.Text_IO;
use Ada.Text_IO;

procedure Record3 is

   type MONTH_NAME is (JAN,FEB,MAR,APR,MAY,JUN,JUL,
                                             AUG,SEP,OCT,NOV,DEC);

   type DATE is
      record
         Month : MONTH_NAME;
         Day   : INTEGER range 1..31;
         Year  : INTEGER range 1776..2010;
      end record;

   type GRADE_ARRAY is array(1..4) of POSITIVE;

   type PERSON is
      record
         Name            : STRING(1..15);
         Birth_Day       : DATE;
         Graduation_Day  : DATE := (MAY,27,1987);
         Age             : INTEGER := 21;
         Sex             : CHARACTER := 'F';
         Grades          : GRADE_ARRAY;
      end record;

   Self, Mother, Father : PERSON;

begin

   Self.Name := "John Q. Doe    ";
   Self.Sex := 'M';
   Self.Birth_Day.Month := OCT;
   Self.Birth_Day.Day := 18;
   Self.Birth_Day.Year := 1938;
   Self.Grades(1) := 85;
   Self.Grades(2) := 90;
   Self.Grades(3) := 75;
   Self.Grades(4) := 92;

   Mother := Self;
   Father.Birth_Day := Mother.Birth_Day;
   Father.Birth_Day.Day := Self.Birth_Day.Day - 4;
   Mother.Sex := 'F';

end Record3;




-- Result of execution

--   (No output from this program.)
```

Examine the file named e_c12_p3.ada and you will find an array type declared in line 17 which is then used in the record type **PERSON**. The addition allows a variable of type **PERSON** to store four grades giving us a little additional flexibility over the last program. The method of assigning data to the new fields are illustrated in lines 38 through 41 and should require no additional comment, because you are already versed on how to use arrays. One rule must be mentioned here, you are not allowed to declare an array with an anonymous type within a record, it must be named. Be sure to compile and run this program.

You will notice that lines 23 through 25 have default values assigned to some elements of the record. The default values will be assigned to those elements every time a record of this type is declared. Of course, the programmer can immediately override the default values by assigning any values he desires, but the default values will be used to initialize those particular members.

AN ARRAY OF RECORDS

Example program ------> e_c12_p4.ada

```
                                        -- Chapter 12 - Program 4
with Ada.Text_IO;
use Ada.Text_IO;

procedure Record4 is

   type MONTH_NAME is (JAN,FEB,MAR,APR,MAY,JUN,JUL,
                                             AUG,SEP,OCT,NOV,DEC);

   type DATE is
      record
         Month : MONTH_NAME;
         Day   : INTEGER range 1..31;
         Year  : INTEGER range 1776..2010;
      end record;

   type PERSON is
      record
         Name      : STRING(1..15);
         Birth_Day : DATE;
         Age       : INTEGER := 15;
         Sex       : CHARACTER := 'M';
      end record;

   Teacher      : PERSON;
   Class_Member : array(1..35) of PERSON;
   Standard     : constant PERSON := ("John Q. Doe    ",
                                       (MAR, 27, 1955), 33, 'M');

   type EMPTY_RECORD is
      record
         null;
      end record;

   type ANOTHER_EMPTY_RECORD is null record;

begin

   Teacher.Name := "John Q. Doe    ";
   Teacher.Age := 21;
   Teacher.Sex := 'M';
   Teacher.Birth_Day.Month := OCT;
   Teacher.Birth_Day.Day := 18;
   Teacher.Birth_Day.Year := 1938;

   for Index in Class_Member'RANGE loop
      Class_Member(Index).Name := "Suzie Lou Q    ";
      Class_Member(Index).Birth_Day.Month := MAY;
      Class_Member(Index).Birth_Day.Day := 23;
      Class_Member(Index).Birth_Day.Year := 1956;
      Class_Member(Index).Sex := 'F';
   end loop;

   Class_Member(4).Name := "Little Johhny  ";
   Class_Member(4).Sex := 'M';
   Class_Member(4).Birth_Day.Day := 17;
   Class_Member(7).Age := 14;
   Class_Member(2) := Standard;
   Class_Member(3) := Standard;

end Record4;




-- Result of execution

--   (No output from this program.)
```

Examine the file named e_c12_p4.ada for an example of an array of records. The types **DATE** and **PERSON** are declared in a manner similar to their declaration in e_c12_p2.ada, but in line 26 we declare an array of 35 variables, each of type **PERSON**, so each is composed of six variable fields. In lines 46 through 52, we assign some nonsense data to each field of the 35 variables by using a loop. Finally, we assign nonsense data to a few of the variables to illustrate how it can be done in lines 54 through 59. You should have no problem understanding this program.

Note that we could have assigned data to one of the records, the first for instance, then used it in a loop to assign values to all of the others by using a record assignment such as, "Class_Member(Index) := Class_Member(1);", and looping from 2 to 35. In a useful program, the data to be assigned will be coming from a file somewhere, as we would probably be filling a database. This is, in fact, the beginning of a very crude database.

Another new construct is illustrated in lines 27 and 28 where we initialize the variable named **Standard** to the aggregate given. Note that, like the unnested record requirement, a nested record must be initialized with an aggregate which includes all values. Lines 30 through 33 illustrate a method of defining a **null** record which will be useful when we study inheritance later in this tutorial. Compile and run this program to assure yourself that it really will compile and run.

It would be good for you to examine section 3.8 of the ARM at this time to become familiar with its language and the method of definition used there.

THE VARIANT RECORD

The variant record is available with Ada 95 and it will be covered in detail in chapter 20, but it is largely superseded by the more powerful techniques of inheritance and type extension which are available with Ada 95.

PROGRAMMING EXERCISES

1. Rewrite e_c12_p4.ada and change the initialization aggregate for Standard from a positional aggregate to a named aggregate.(Solution)

```
                               -- Chapter 12 - Programming exercise 1
with Ada.Text_IO;
use Ada.Text_IO;

procedure CH12_1 is

   type MONTH_NAME is (JAN,FEB,MAR,APR,MAY,JUN,JUL,
                                                 AUG,SEP,OCT,NOV,DEC);

   type DATE is
      record
         Month : MONTH_NAME;
         Day   : INTEGER range 1..31;
         Year  : INTEGER range 1776..2010;
      end record;

   type PERSON is
      record
         Name      : STRING(1..15);
         Birth_Day : DATE;
         Age       : INTEGER := 15;
         Sex       : CHARACTER := 'M';
      end record;

   Teacher      : PERSON;
   Class_Member : array(1..35) of PERSON;
   Standard     : constant PERSON :=
                        (Birth_Day => (Month => MAR,
                                       Year => 1955,
                                       Day => 27),
                         Name => "John Q. Doe    ",
                         Age => 33,
                         Sex => 'M');

   type EMPTY_RECORD is
      record
         null;
      end record;

begin

   Teacher.Name := "John Q. Doe    ";
   Teacher.Age := 21;
   Teacher.Sex := 'M';
   Teacher.Birth_Day.Month := OCT;
   Teacher.Birth_Day.Day := 18;
   Teacher.Birth_Day.Year := 1938;

   for Index in Class_Member'RANGE loop
      Class_Member(Index).Name := "Suzie Lou Q    ";
      Class_Member(Index).Birth_Day.Month := MAY;
      Class_Member(Index).Birth_Day.Day := 23;
      Class_Member(Index).Birth_Day.Year := 1956;
      Class_Member(Index).Sex := 'F';
   end loop;

   Class_Member(4).Name := "Little Johhny  ";
   Class_Member(4).Sex := 'M';
   Class_Member(4).Birth_Day.Day := 17;
   Class_Member(7).Age := 14;
   Class_Member(2) := Standard;
   Class_Member(3) := Standard;

end CH12_1;




-- Result of execution
--   (No output from this program.)
```

2. Rewrite e_c12_p1.ada to utilize the enumerated type for the month field as used in e_c12_p3.ada. Note that you will need to instantiate a copy of the package named **Enumerated_IO** to display the month name.(Solution)

```
                                       -- Chapter 12 - Program 1
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure CH12_2 is

   type MONTH_NAME is (JAN,FEB,MAR,APR,MAY,JUN,JUL,
                                             AUG,SEP,OCT,NOV,DEC);

   package Enum_IO is new Ada.Text_IO.Enumeration_IO(MONTH_NAME);
   use Enum_IO;

   type DATE is
      record
         Month : MONTH_NAME;
         Day   : INTEGER range 1..31;
         Year  : INTEGER range 1776..2010;
      end record;

   Independence_Day : DATE;
   Birth_Day        : DATE;
   Today,Pay_Day    : DATE := (MAY,25,1982);

begin
   Independence_Day.Month := JUL;
   Independence_Day.Day := 4;
   Independence_Day.Year := 1776;

   Birth_Day := Independence_Day;

   Pay_Day.Day := 30;

   Put("Independence day was on ");
   Put(Independence_Day.Month, 2);
   Put(Independence_Day.Day, 2);
   Put(",");
   Put(Independence_Day.Year, 5);
   New_Line;

   Birth_Day := (Day => 19, Month => FEB, Year => 1937);
   Today := (JUL, 14, 1952);
   Pay_Day := (JUL, Year => 1954, Day => 17);

end CH12_2;




-- Result of execution
```

-- Independence day was on JUL 4, 1776

Ada Tutorial - Chapter 13

THE ACCESS TYPE VARIABLE

THE ACCESS TYPE IS DIFFERENT

Example program ------> **e_c13_p1.ada**

```
                                       -- Chapter 13 - Program 1
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure Access1 is

   type POINT_SOMEWHERE is access INTEGER;
   Index, Arrow, There : POINT_SOMEWHERE;

begin
   Index := new INTEGER;
   Index.all := 13;
   Put("The value is");
   Put(Index.all, 6);
   New_Line;

   Arrow := new INTEGER;
   Arrow.all := Index.all + 16;
   There := Arrow;
   Put("The values are now");
   Put(Index.all, 6);
   Put(Arrow.all, 6);
   Put(There.all, 6);
   New_Line;

   There.all := 21;
   Put("The values are now");
   Put(Index.all, 6);
   Put(Arrow.all, 6);
   Put(There.all, 6);
   New_Line;

end Access1;




-- Result of execution

-- The value is    13
-- The values are now    13    29    29
-- The values are now    13    21    21
```

The access type variable is different from every other type we have encountered because it is not actually a variable which can store a piece of data, but contains the address of another piece of data which can be manipulated. As always the best teacher is an example, so examine the file named e_c13_p1.ada for an example program with a few access type variables in it.

DECLARING AN ACCESS TYPE VARIABLE

In line 7 we declare a new type, an **access** type. As with all types, the reserved word **type** is given, followed by the type name, then the reserved words **is** and the type definition. The type definition begins with the reserved word **access**, which denotes an access type variable, then by the type which we wish to access. The type which we wish to access can be any type which has been declared prior to this point in the program, either a predeclared type or a type we have declared. Because there are no predeclared access types available in Ada, we must declare all access types we wish to use.

The new type is used to declare three access variables in line 8. No actual variables are declared, only access to three places in memory which actually do not exist yet. The access type variables do not store an integer value, as might be expected, but instead store the address of an integer value located somewhere within the address space of the computer memory. Note that the three access variables are automatically initialized to **null** when they are declared, as are all access variables in Ada.

Figure 13-1 illustrates graphically the condition of the system at this point. A box with a dot in the center depicts an access variable and an empty box will be used to depict a scalar variable.

WE NEED SOME DATA TO POINT AT

As we begin the executable part of the program, we have no data to access, so we create some data storage in line 11 using the **new** reserved word. This tells the system to go somewhere and create a variable of type **INTEGER**, with no name, and cause the access variable named **Index** to point at this new variable. The effective address of the new variable is assigned to the access variable **Index**, but the new variable still has no assigned value.

Line 12 tells the system to assign the value of 13 to the new variable by using a very strange looking method of doing so. For the first two example programs in this chapter, we will simply say that the value of all of the variable is set to the indicated value, namely 13. The end result is that the access variable named **Index** is pointing someplace in memory which has no name, but contains the value of 13. Figure 13-2 illustrates our current situation.

Lines 13 through 15 indicate that it is possible to display the value of this variable using the same method we have used in all earlier lessons of this tutorial. If you remember to add the **.all** to the access variable, you will be referring to the data stored at the location which it accesses.

Line 17 is used to create another variable of type **INTEGER** somewhere in memory, with **Arrow** pointing to it, but which contains no value as yet. The next line says to take the value that is stored at the location to which **Index** points, add 16 to it, and store the result, which should be 29, in the location to which **Arrow** points. **Arrow** actually "accesses" the variable, but it is a bit more descriptive to use the word points, especially if you are a Pascal or C programmer.

THE THIRD ACCESS VARIABLE

We have not yet used the access variable named **There**, so we instruct the system, in line 19, to cause it to point to the same piece of data which **Arrow** is currently accessing. By failing to add the **all** to the two access variables, we are assigning the **access** address to **There** rather than the value which **Arrow** accesses. If only one of them had the **all** appended in line 19, there would be a type clash resulting in a compiler error message. All three access variables are accessing some data somewhere, so we can use their **.all** notation to display all three values. See figure 13-3 for a graphic representation of the current data space.

Note that there are actually only two variables, because two of the access variables are pointing at the same piece of data. This is illustrated when one of the variables is changed in line 26, and when the data is printed, it is evident that two of the values are different.

Be sure to compile and run this program. When you think you understand it, see if you can modify it such that all three access variables point to the same piece of data.

ACCESSING INTEGER AND FLOAT TYPE VARIABLES

Example program ------> **e_c13_p2.ada**

```
                                       -- Chapter 13 - Program 2
with Ada.Text_IO, Ada.Integer_Text_IO, Ada.Float_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO, Ada.Float_Text_IO;

procedure Access2 is

   type POINT_TO_INT is access INTEGER;
   Index, Arrow : POINT_TO_INT;

   type POINT_TO_FLOAT is access FLOAT;
   X, Y, Z : POINT_TO_FLOAT;

begin
   Index := new INTEGER'(173);
   Arrow := new INTEGER'(57);
   Put("The values are");
   Put(Index.all, 6);
   Put(Arrow.all, 6);
   New_Line;
   Index.all := 13;
   Arrow.all := Index.all;
   Index := Arrow;

   X := new FLOAT'(3.14159);
   Y := X;
   Z := X;
   Put("The float values are");
   Put(X.all, 6, 6, 0);
   Put(Y.all, 6, 6, 0);
   Put(Z.all, 6, 6, 0);
   New_Line;

   X.all := 2.0 * Y.all;
   Put("The float values are");
   Put(X.all, 6, 6, 0);
   Put(Y.all, 6, 6, 0);
   Put(Z.all, 6, 6, 0);
   New_Line;

end Access2;
```

```
-- Result of execution

-- The values are   173    57
-- The float values are     3.141590     3.141590     3.141590
-- The float values are     6.283180     6.283180     6.283180
```

Examine the file named e_c13_p2.ada for some additional examples of access type variables. We begin by declaring two access variables which access **INTEGER** type variables and three access variables that access **FLOAT** type variables. It should be pointed out, and it probably comes as no surprise to you, that it is illegal to attempt to access a variable with the wrong type of access variable. Explicit type conversion is possible concerning the data types, but not the access types.

Line 14 introduces a new construct, that of initializing a variable when it is created. Using a form similar to qualification, an **INTEGER** type variable is created somewhere in memory, initialized to 173, and the access variable named **Index** is assigned its address so that it points to it, or accesses it. After executing line 15, the data space is as shown in figure 13-4.

Be sure to note the difference between the expressions in lines 21 and 22. In line 21, the value stored at the place where **Index** points, is stored at the place where **Arrow** points. However, in line 22, the access variable **Index** is caused to point to the same location where the access variable **Arrow** points.

A FLOAT TYPE ACCESS VARIABLE

Line 24 illustrates creation of a **FLOAT** type variable initialized with the value of Pi. Since the access variable names are used in lines 25 and 26 without the **all** appended, all three **FLOAT** type access variables are assigned to access the same variable, and some results are displayed. Figure 13-5 illustrates the condition of the system at this point.

The single **FLOAT** type variable is doubled in line 33, and it is displayed again three different ways. Be sure to compile and execute this program.

ACCESSING A RECORD VARIABLE

Example program ------> **e_c13_p3.ada**

```
                                       -- Chapter 13 - Program 3
with Ada.Text_IO, Ada.Integer_Text_IO, Ada.Unchecked_Deallocation;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure Access3 is
```

```
   type MY_RECORD is
      record
         Age     : INTEGER;
         Initial : CHARACTER;
         Sex     : CHARACTER;
      end record;

   type ACCESS_MY_DATA is access MY_RECORD;

   procedure Free is new
             Ada.Unchecked_Deallocation(MY_RECORD, ACCESS_MY_DATA);

   Myself  : ACCESS_MY_DATA;
   Friend  : ACCESS_MY_DATA := new MY_RECORD'(30, 'R', 'F');

   Result : BOOLEAN;

begin

   Myself := new MY_RECORD;

   Myself.Age := 34;
   Myself.Initial := 'D';
   Myself.Sex := 'M';

   Friend := new MY_RECORD'(31, 'R', 'F');

   Put("My age is");
   Put(Myself.Age, 3);
   Put(" and my initial is ");
   Put(Myself.Initial);
   New_Line;

   Friend.all := Myself.all;

   Result := Friend.all = Myself.all;    -- TRUE because of line 43
   Result := Friend = Myself;            -- FALSE because they point
                                         -- to different things.

   Free(Myself);
   Free(Friend);

end Access3;




-- Result of execution

-- My age is 34 and my initial is D
```

Examine the example program named e_c13_p3.ada for some additional uses for access variables. This program begins by defining a record type, then an **access** type which can be used to access data of this record type. Ignore the procedure **Free** in line 16 for a short time. In line 19, we declare a variable named **Myself** which is an access variable that accesses a variable of the type **MY_RECORD**. Since the record does not exist yet, the access variable is actually pointing nowhere. According to the Ada definition, the created access variable will be initialized to the value **null**, which means it points nowhere. This value can be tested for some value as we shall see later. All access variables used in an Ada program, regardless of how they are declared, will be initially assigned the value of **null**. This is true, unless they are specifically initialized to some value as shown in line 20.

Line 20 is very interesting because we declare an access variable named **Friend**, and initialize the access variable by creating a new record somewhere in memory, then initialize the record itself to the values given in the positional aggregate. Finally, the access variable **Friend** is caused to point to the newly created record. It is permissible to create a new record, but omit the initialization, supplying the initial values in the executable part of the program as we are doing with the variable named **Myself**. We finally declare a **BOOLEAN** type variable for later use. Figure 13-6 illustrates our current data space.

USING THE RECORD ACCESS VARIABLE

In line 26, we create a new variable of type **MY_RECORD**, which is composed of three separate fields. The three fields are assigned in much the same manner that they were assigned in the chapter where we studied records, so this should pose no problem for you. In line 32, we create another new record somewhere and initialize it to the values given, and cause the variable named **Friend** to point to it, or access it. See figure 13-7.

NOW WE HAVE SOME LOST VARIABLES

Consider that the access variable **Friend** already had some data that it was pointing at, and we told the system to cause it to point at this newly created record. The record it was formerly pointing at is now somewhere in memory, but has nothing pointing at it, so it is in effect lost. We cannot store anything in it, nor can we read out the data that is stored in it. Of even more consequence, we cannot free up those memory locations for further use, so the space is totally lost to our program. Of course, the operating system will take care of cleaning up all of the lost variables when our program terminates, so the data is not lost forever. It is up to us to see that we do not lose memory space through clumsy programming because Ada cannot check to see that we have reassigned an access variable.

The next interesting thing is illustrated in line 40 where **all** of the fields of the record which **Myself** accesses are assigned to all of the fields of the record which **Friend** accesses. Now it makes sense why the designers of Ada chose to refer to the data which is accessed by the **.all** notation, it refers to all of the data that the access variable points to. It should be pointed out that as you gain experience with Ada you will find that nearly all access type variables are used to access records, and few, if any, will access scalar variables.

BOOLEAN OPERATIONS WITH ACCESS VARIABLES

Records accessed by access variables can be compared for equality or inequality, just like regular records, and they are equal only if all fields in one record are equal to the corresponding fields in the other record. Line 42 will therefore result in **TRUE**, because the records are identical, due to the assignment in line 40. Access variables can also be compared to each other, and result in **TRUE** only if they are both pointing to the same object. In line 43, the result is **FALSE** because they are pointing to different records, even though the records they point to happen to be equal to each other.

WHAT IS UNCHECKED DEALLOCATION?

The procedure **Unchecked_Deallocation** is a required part of Ada, so your compiler writer has supplied you with this procedure as a part of the library. Any dynamically allocated data can be freed up for reuse by the system through use of this procedure as illustrated in this program. You must first instantiate a copy of the generic procedure as illustrated in line 16, and name it any available identifier you choose. You must supply two types as parameters, the first being the object type you wish to deallocate, and the second being the access type. The name **Free** is generally used because that name is used for the equivalent procedure in Pascal and in C.

To actually deallocate some storage, you use the name of the access variable which is accessing the storage to be released as the only parameter of the procedure named **Free** as illustrated in lines 46 and 47. This storage is then available for reuse by some other part of the program.

There is a lot more to be said about deallocation of storage and the way it is accomplished, but the details will be left until chapter 23, after you gain more experience with Ada. Until then, with your limited knowledge of Ada, you will probably not be writing programs in which you will need this information. Be sure to compile and execute this program.

GARBAGE COLLECTION

Another way to deallocate the data accessed by the access variables, is to assign the value of **null** to the access variables. This will cause the dynamically allocated variables to have no access variables accessing them, and they are then unusable, or garbage. An Ada implementation may implement a garbage collector to search for un-accessed data and reclaim the storage for further use. Implementation of a garbage collector is optional according to the ARM. Much more will be said about deallocation and garbage collection in chapter 25.

ACCESSING AN ARRAY OF DATA

Example program ------> e_c13_p4.ada

```
                                       -- Chapter 13 - Program 4
with Ada.Text_IO, Ada.Integer_Text_IO, Ada.Unchecked_Deallocation;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure Access4 is

   type MY_ARRAY is array(3..8) of INTEGER;
   type POINT_TO_ARRAY is access MY_ARRAY;

   procedure Free is new
             Ada.Unchecked_Deallocation(MY_ARRAY, POINT_TO_ARRAY);

   List_Of_Stuff : MY_ARRAY := (34, 12, -14, 1, 27, -11);
   There         : POINT_TO_ARRAY;
   Here          : POINT_TO_ARRAY;

begin
```

```
   There := new MY_ARRAY;
   There.all := List_Of_Stuff;
   Here := There;

   for Index in MY_ARRAY'RANGE loop
      Put(List_Of_Stuff(Index), 6);
      Put(There.all(Index), 6);
      Put(Here.all(Index), 6);
      New_Line;
   end loop;

   Free(There);

end Access4;




-- Result of execution

--      34    34    34
--      12    12    12
--     -14   -14   -14
--       1     1     1
--      27    27    27
--     -11   -11   -11
```

The example program named e_c13_p4.ada gives an example of using an access variable to access an array. The only thing that could be considered new here is the assignment in line 20 where the value of the array **List_Of_Stuff**, is assigned to the variable which is accessed by the access variable **There**. Note that 6 **INTEGER** type values are actually assigned in this one statement.

Note that **Unchecked_Deallocation** is illustrated here also as an example. The program should be simple for you to follow, and after you understand it, compile and execute it.

AN ARRAY OF ACCESS VARIABLES

Example program ------> **e_c13_p5.ada**

```
                                       -- Chapter 13 - Program 5
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure Access5 is

   type MY_RECORD is
      record
         Age     : INTEGER;
         Initial : CHARACTER;
         Sex     : CHARACTER;
      end record;

   type ACCESS_MY_DATA is access MY_RECORD;

   Myself  : ACCESS_MY_DATA;
   Class   : array(1..10) of ACCESS_MY_DATA;

begin
   Myself := new MY_RECORD;
   Myself.Age := 34;
   Myself.Initial := 'D';
   Myself.Sex := 'M';

   for Index in 1..10 loop
      Class(Index) := new MY_RECORD;
      Class(Index).all := Myself.all;
   end loop;

   Class(3).Age := 30;
   Class(3).Initial := 'A';
   Class(5).Initial := 'Z';
   Class(8).Initial := 'R';
   Class(6).Sex := 'F';
   Class(7).Sex := 'F';
   Class(2).Sex := 'F';

   for Index in 1..10 loop
      Put("The class members age is");
      Put(Class(Index).Age, 3);
      if Class(Index).Sex = 'M' then
         Put(" and his initial is ");
      else
         Put(" and her initial is ");
      end if;
      Put(Class(Index).Initial);
      New_Line;
   end loop;

end Access5;




-- Result of execution

-- The class members age is 34 and his initial is D
-- The class members age is 34 and her initial is D
-- The class members age is 30 and his initial is A
-- The class members age is 34 and his initial is D
-- The class members age is 34 and his initial is Z
-- The class members age is 34 and her initial is D
-- The class members age is 34 and her initial is D
-- The class members age is 34 and his initial is R
-- The class members age is 34 and his initial is D
-- The class members age is 34 and his initial is D
```

Examine the program e_c13_p5.ada for an example including an array of access variables which is declared in line 17. The variable named **Class** is composed of a total of ten access type variables, any of which can be used to point to a variable of type **MY_RECORD.** The loop in lines 26 through 29 is used to first create a record variable, then assign values to the fields of the created record variable, for each of the ten access variables. Since each record is composed of three subfields, a total of 30 separate variables are created and assigned values in this loop. A few of the variables are reassigned values in lines 31 through 37 for illustrative purposes, and the entire array named **Class** is displayed on the monitor.

Note that the array declared in line 17 is an anonymous type array.

Compile and run this program, and be sure you understand the resulting printout. The access variable will be very important when we study some of the advanced programming techniques later in this tutorial.

ACCESS TO STATIC OBJECTS

Example program ------> **e_c13_p6.ada**

```
                                            -- Chapter 13 - Program 6
with Ada.Text_IO, Ada.Integer_Text_IO, Ada.Float_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO, Ada.Float_Text_IO;

procedure AcessAll is

   type BRICK_TYPE is record
      Length : INTEGER;
      Width  : INTEGER;
      Height : INTEGER;
   end record;

   type ACCESS_INT   is access all INTEGER;
   type ACCESS_FLT   is access all FLOAT;
   type ACCESS_BRICK is access all BRICK_TYPE;

   Number   : INTEGER            := 27;
   Count    : aliased INTEGER    := 12;
   Pt_Count : ACCESS_INT;

   Size     : aliased FLOAT      := 7.4;
   Pt_Size  : ACCESS_FLT;

   Brick    : aliased BRICK_TYPE := (3, 4, 6);
   Pt_Brick : ACCESS_BRICK;

begin

   Pt_Count := Count'Access;
-- Pt_Count := Number'Access;  Illegal - Number not aliased
-- Pt_Count := Size'Access;    Illegal - Size is FLOAT
-- Pt_Count := Brick'Access;   Illegal - Brick is BRICK_TYPE
   Pt_Count.All := Pt_Count.All + 13;
   Put("The value of Count is now ");
   Put(Count, 5);
   New_Line;

   Pt_Size := Size'Access;
   Put("The value of Size is now ");
   Put(Pt_Size.All, 5, 2, 0);
   New_Line;

   Pt_Brick := Brick'Access;
   Put("The brick measures  ");
   Put(Pt_Brick.Length, 5);   -- Using the pointer
   Put(Pt_Brick.Width, 5);     -- Using the pointer
   Put(Brick.Height, 5);       -- Using the object
   New_Line;

end AcessAll;



-- Result of execution
--
-- The value of Count is now    25
-- The value of Size is now     7.40
-- The brick measures      3    4    6
```

Ada 95 has the ability to access a local or global variable with an access variable. This was not permitted with Ada 83. In order to do so, it is necessary to identify the access variable as one with the ability to access a local or global variable by using the keyword **all** as is done in lines 13 through 15 of the example program. In addition, the variables to be accessed must be identified as accessible with the new keyword **aliased** as illustrated in lines 18, 21, and 24 of this program. This is a practice that could be easily abused, so the designers of Ada force you to identify the access variable and the variable to be accessed prior to actually doing the operation. This forces you to think carefully about what you are doing. The use and abuse of pointers is the weakest spot in some other languages and lead to very difficult to find errors.

You will note that the Ada compiler will complain, via an error, if any of the ingredients is missing while using this technique. Lines 30 through 32 are included in this program to illustrate the various rules of usage.

There is one other rule that must be followed and is checked by the compiler anytime you use an access variable to refer to a local or global variable. The object referred to must exist for as long or longer than the access variable itself, or the compiler will issue an error message. This is to prevent the error of using an access variable when the object it refers to is no longer in existence.

ACCESS TO SUBPROGRAMS

Example program ------> e_c13_p7.ada

```
                                                -- Chapter 13 - Program 7
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure AcessFnc is

   function Double_It(In_Value : INTEGER) return INTEGER is
   begin
      return 2 * In_Value;
   end Double_It;

   function Triple_It(In_Value : INTEGER) return INTEGER is
   begin
      return 3 * In_Value;
   end Triple_It;

   function Decade_It(Number   : INTEGER) return INTEGER is
   begin
      return 10 * Number;
   end Decade_It;

   type ACCESS_FUNCTION is access function(X : INTEGER) return INTEGER;
   Multiply : ACCESS_FUNCTION;

   Worker : INTEGER := 17;

begin

   Multiply := Double_It'Access;
   Put("Double value is");
   Put(Multiply(Worker), 6);
   New_Line;

   Multiply := Decade_It'Access;
   Put("Ten times value is");
   Put(Multiply(Worker), 6);
   New_Line;

   Multiply := Triple_It'Access;
   Put("Triple value is");
   Put(Multiply(Worker), 6);
   New_Line;

end AcessFnc;




-- Result of execution
--
-- Double value is    34
-- Ten times value is   170
-- Triple value is    51
```

The example program named e_c13_p7.ada is an example of using a single access variable to call three different functions. There are three functions defined in lines 7 through 20, and each have the same number of parameters, the same type of parameters and the same return type. With all of this parallelism, they are candidates for use with an access variable. In line 22 we define a type which happens to have the same signature as the three functions listed above it, and finally we declare an access variable of the new type named **Multiply**. In the executable part of the program we call the three functions with the access variable in lines 31, 36, and 41. The same call results in calls to different physical functions since it is pointing to a different function each time we call it. Lines 29, 34, and 39 are where we actually assign the address to the access variable.

This same technique can be used with procedure calls as well, provided that the signature for each of the entities is identical. In the case of the procedure, all of the parameters must have the same mode. They are by definition, all of the **in** mode in a function.

There is one other rule that must be followed and is checked by the compiler anytime you use an access variable to refer to a subprogram. The object referred to must exist for as long or longer than the access variable itself, or the compiler will issue an error message. This is to prevent the use of an access variable when the object it refers to is no longer in existence, an obvious error. In those cases when you think it is all right to override this rule, **Unchecked_Access** is available for use and documented in the ARM.

The use of anything that is unchecked is highly discouraged.

PROGRAMMING EXERCISES

1. Declare a record named **BOX_TYPE** composed of three **FLOAT** type elements, **Length**, **Width**, and **Height**. Use the **new** function to create three boxes named **Small**, **Medium**, and **Large**, and store suitable values in all nine fields. Display the data concerning the three boxes along with the volume of each box.(Solution)

```
                                    -- Chapter 13 - Programming exercise 1
with Ada.Text_IO, Ada.Float_Text_IO;
use Ada.Text_IO, Ada.Float_Text_IO;

procedure Ch13_1 is

   type BOX_TYPE is
      record
         Length : FLOAT;
         Width  : FLOAT;
         Height : FLOAT;
      end record;
   type POINT_TO_BOX is access BOX_TYPE;

   Small, Medium, Large : POINT_TO_BOX;

begin

   Small  := new BOX_TYPE;
   Medium := new BOX_TYPE;
   Large  := new BOX_TYPE;

   Small.Length := 3.5;
   Small.Width  := 2.4;
   Small.Height := 1.9;
   Medium.Length := 7.4;
   Medium.Width  := 6.4;
   Medium.Height := 9.3;
   Large.Length := 13.8;
   Large.Width  := 21.5;
   Large.Height := 15.1;

   Put(Small.Length, 8, 3, 0);
   Put(Small.Width, 8, 3, 0);
   Put(Small.Height, 8, 3, 0);
   Put(Small.Length * Small.Width * Small.Height, 8, 3, 0);
   New_Line;

   Put(Medium.Length, 8, 3, 0);
   Put(Medium.Width, 8, 3, 0);
   Put(Medium.Height, 8, 3, 0);
   Put(Medium.Length * Medium.Width * Medium.Height, 8, 3, 0);
   New_Line;

   Put(Large.Length, 8, 3, 0);
   Put(Large.Width, 8, 3, 0);
   Put(Large.Height, 8, 3, 0);
   Put(Large.Length * Large.Width * Large.Height, 8, 3, 0);
   New_Line;

end Ch13_1;




-- Result of execution

--        3.500       2.400       1.900      15.960
--        7.400       6.400       9.300     440.448
--       13.800      21.500      15.100    4480.170
```

2. Add the **Unchecked_Deallocation** procedure to the other three example programs, e_c13_p1.ada, e_c13_p2.ada, and e_c13_p5.ada, and deallocate all allocated variables. (Solution 1)(Solution 2)(Solution 3)

```
                           -- Chapter 13 - Programming Exercise 2
                                        -- Chapter 13 - Program 1
with Ada.Text_IO, Ada.Integer_Text_IO, Unchecked_Deallocation;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure Ch13_2a is
```

```
   type POINT_SOMEWHERE is access INTEGER;
   Index,Arrow,There : POINT_SOMEWHERE;

   procedure Free is new
                Unchecked_Deallocation(INTEGER, POINT_SOMEWHERE);

begin
   Index := new INTEGER;
   Index.all := 13;
   Put("The value is");
   Put(Index.all, 5);
   New_Line;

   Arrow := new INTEGER;
   Arrow.all := Index.all + 16;
   There := Arrow;
   Put("The values are now");
   Put(Index.all, 5);
   Put(Arrow.all, 5);
   Put(There.all, 5);
   New_Line;

   There.all := 21;
   Put("The values are now");
   Put(Index.all, 5);
   Put(Arrow.all, 5);
   Put(There.all, 5);
   New_Line;

   Free(Index);
   Free(Arrow);

end Ch13_2a;




-- Result of execution

-- The value is   13
-- The values are now   13   29   29
-- The values are now   13   21   21

-- Note that the Free procedure in line 41 could have been done
--   with the access variable There since it is also accessing
--   that data.  Both could not be used however.


                          -- Chapter 13 - Programming Exercise 2
                                       -- Chapter 13 - Program 2
with Ada.Text_IO, Unchecked_Deallocation;
use Ada.Text_IO;

procedure Ch13_2b is

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

   type POINT_TO_INT is access INTEGER;
   Index,Arrow : POINT_TO_INT;
   type POINT_TO_FLOAT is access FLOAT;
   X,Y,Z : POINT_TO_FLOAT;

   procedure Free is new
                   Unchecked_Deallocation(INTEGER, POINT_TO_INT);
   procedure Free is new
                    Unchecked_Deallocation(FLOAT,POINT_TO_FLOAT);

begin
   Index := new INTEGER'(173);
   Arrow := new INTEGER'(57);
   Put("The values are");
   Put(Index.all, 6);
   Put(Arrow.all, 6);
   New_Line;
   Index.all := 13;
   Arrow.all := Index.all;
   Index := Arrow;

   X := new FLOAT'(3.14159);
   Y := X;
   Z := X;
   Put("The float values are");
   Put(X.all, 6, 6, 0);
   Put(Y.all, 6, 6, 0);
   Put(Z.all, 6, 6, 0);
   New_Line;

   X.all := 2.0 * Y.all;
   Put("The float values are");
   Put(X.all, 6, 6, 0);
   Put(Y.all, 6, 6, 0);
   Put(Z.all, 6, 6, 0);
   New_Line;

   Free(Index);
   Free(X);

end Ch13_2b;




-- Result of execution

-- The values are   173    57
-- The float values are     3.141590     3.141590     3.141590
-- The float values are     6.283180     6.283180     6.283180

-- Note that the deallocations could be done with any of the
--  variable names since all variables of the same type are point-
--  ing to the same places.

-- Note also that the two procedures could be named something else
--  rather than overloading the name Free.  It would be better
--  practice to use different names for the variables.



                           -- Chapter 13 - Programming Exercise 2
                                        -- Chapter 13 - Program 5
```

```
with Ada.Text_IO, Ada.Integer_Text_IO, Unchecked_Deallocation;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure Ch13_2c is

   type MY_RECORD is
      record
         Age     : INTEGER;
         Initial : CHARACTER;
         Sex     : CHARACTER;
      end record;

   type ACCESS_MY_DATA is access MY_RECORD;

   procedure Free is new
            Unchecked_Deallocation(MY_RECORD,ACCESS_MY_DATA);

   Myself  : ACCESS_MY_DATA;
   Class   : array(1..10) of ACCESS_MY_DATA;

begin
   Myself := new MY_RECORD;

   Myself.Age := 34;
   Myself.Initial := 'D';
   Myself.Sex := 'M';

   for Index in 1..10 loop
      Class(Index) := new MY_RECORD;
      Class(Index).all := Myself.all;
   end loop;

   Class(3).Age := 30;
   Class(3).Initial := 'A';
   Class(5).Initial := 'Z';
   Class(8).Initial := 'R';
   Class(6).Sex := 'F';
   Class(7).Sex := 'F';
   Class(2).Sex := 'F';

   for Index in 1..10 loop
      Put("The class members age is");
      Put(Class(Index).Age, 3);
      if Class(Index).Sex = 'M' then
         Put(" and his initial is ");
      else
         Put(" and her initial is ");
      end if;
      Put(Class(Index).Initial);
      New_Line;
   end loop;

   for Index in 1..10 loop
      Free(Class(Index));
   end loop;
   Free(Myself);

end Ch13_2c;
```

```
-- Result of execution

-- The class members age is 34 and his initial is D
-- The class members age is 34 and her initial is D
-- The class members age is 30 and his initial is A
-- The class members age is 34 and his initial is D
-- The class members age is 34 and his initial is Z
-- The class members age is 34 and her initial is D
-- The class members age is 34 and her initial is D
-- The class members age is 34 and his initial is R
-- The class members age is 34 and his initial is D
-- The class members age is 34 and his initial is D
```

Ada Tutorial - Chapter 14

TEXT INPUT/OUTPUT

ADA IS NOT BIG ON I/O

Ada was originally designed as a language for embedded real-time systems rather than as a business oriented language, so input and output of data is a rather low priority part of the language. It may surprise you to know that there are no input or output instructions available for use within the Ada language itself, the problem being left to the compiler writers. However, the designers of Ada realized that if the entire topic of input and output were left to the individual compiler writers, users would be left with a mess trying to use nonstandard methods of input and output. They exercised tremendous foresight by defining a standard external library to be used for input and output of data that must be available with all Ada compilers that pass the validation suite.

Even though the libraries are defined, the details of their actions are not precisely defined, so there is still some chance that your compiler will not behave in exactly the same way as the descriptions given here. You will have to check the documentation packaged with your compiler to discover the exact details of input and output operation. You may rest assured however, that the differences will only be in the smallest details.

PACKAGE INSTANTIATION

Before we look at some text output procedures, we need to just scratch the surface on another topic. We mentioned package instantiation several chapters back and we have reached a point where we should define what that is to some extent. Ada provides the ability to write generic packages which can be used to provide the same functionality for several different types without rewriting the code. A generic package does not have a type associated with it, so we tell the system we would like it to work with a certain type by instantiating a copy of the generic package with our particular type. If we have, for example, 4 different types we need to work with, we instantiate a copy for each of the four types and we can then perform the same operations on variables of each of those types. An instantiation is a fancy way of saying we make an "instance" of the generic package. This will all be covered in detail in part 2 of this Ada tutorial.

The Ada 95 environment provides a few of the generic packages pre-instantiated for us. For example, a copy of the generic package **Ada.Text_IO.Integer_IO** is preinstantiated for the type **INTEGER** and named **Ada.Integer_Text_IO** which we have been using in this tutorial. Likewise **Ada.Float_Text_IO** is an instantiation of the generic package named **Ada.Text_IO.Float_IO** for use with the type **FLOAT**. Now we have another problem since we need to define where the generic packages mentioned in this paragraph came from. The best answer is that they came with our compiler, and we will simply use them as illustrated until we study generic packages in part 2 of this tutorial.

There are probably other pre-instantiated packages provided for us by our compiler writers. The types **INTEGER** and **FLOAT** are required to be provided by every Ada compiler, but there other types that can be provided. For example, type **LONG_FLOAT** may be provided as well as **SHORT_FLOAT**, each with its own characteristics, and each with its own input/output package named **Ada.Long_Float_Text_IO** or **Ada.Short_Float_Text_IO**. Other types may be **SHORT_INTEGER**, **SHORT_SHORT_INTEGER**, or other variations of **INTEGER**, with the packages **Ada.Short_Integer_Text_IO** or **Ada.Short_Short_Integer_Text_IO**. You should spend some time studying the capabilities and limitations of your compiler to see just what types are available for your use.

You will notice that many of these packages start with **Ada.** tacked onto the front of the full name. Ada 95 includes the ability to define and use hierarchical libraries and many of the Ada 95 packages are declared as a part of the Ada library. We will cover libraries in more detail later along with the reasons for packaging them in this way.

Following that brief excursion, let's look at some text output procedures.

FORMATTED OUTPUT DATA

Example program ------> e_c14_p1.ada

```
                                       -- Chapter 14 - Program 1
with Ada.Text_IO;
use Ada.Text_IO;

procedure Formats is

   type MY_FIXED is delta 0.01 range 20.0..42.0;
   type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);
   type MY_INTEGER is range -13..323;

   X_Value : FLOAT := 3.14;
   Index   : INTEGER := 27;
   Count   : MY_INTEGER := -7;
   What    : BOOLEAN := TRUE;
   Who     : BOOLEAN := FALSE;
   Size    : MY_FIXED := 24.33;
   Today   : DAY := TUE;

   package Int_IO is new Ada.Text_IO.Integer_IO(INTEGER);
   use Int_IO;
   package Flt_IO is new Ada.Text_IO.Float_IO(FLOAT);
   use Flt_IO;
   package Enum_IO is new Ada.Text_IO.Enumeration_IO(BOOLEAN);
   use Enum_IO;

   package Fix_IO is new Ada.Text_IO.Fixed_IO(MY_FIXED);
   use Fix_IO;
   package Day_IO is new Ada.Text_IO.Enumeration_IO(DAY);
   use Day_IO;
   package New_Int_IO is new Ada.Text_IO.Integer_IO(MY_INTEGER);
   use New_Int_IO;

begin
                                       -- INTEGER outputs
   Put("Index is --->"); Put(Index);   Put("<---"); New_Line;
   Put("Index is --->"); Put(Index,3); Put("<---"); New_Line;
   Put("Index is --->"); Put(Index,8); Put("<---"); New_Line(2);
   Put("Count is --->"); Put(Count);   Put("<---"); New_Line;
   Put("Count is --->"); Put(Count,3); Put("<---"); New_Line;
   Put("Count is --->"); Put(Count,8); Put("<---"); New_Line(2);

                                       -- FLOAT outputs
   Put("Put(X_Value) -------->"); Put(X_Value);        New_Line;
   Put("Put(X_Value,5) ------>"); Put(X_Value,5);      New_Line;
   Put("Put(X_Value,5,5) ---->"); Put(X_Value,5,5);    New_Line;
   Put("Put(X_Value,5,5,0) -->"); Put(X_Value,5,5,0);  New_Line(2);

                                       -- MY_FIXED outputs
   Put("Put(Size) -------->"); Put(Size);        New_Line;
   Put("Put(Size,5) ------>"); Put(Size,5);      New_Line;
   Put("Put(Size,5,5) ---->"); Put(Size,5,5);    New_Line;
   Put("Put(Size,5,5,0) -->"); Put(Size,5,5,0);  New_Line(2);

                                       -- BOOLEAN outputs
   Put("What is ---->"); Put(What);   Put("<---"); New_Line;
   Put("Who is ----->"); Put(Who);    Put("<---"); New_Line;
   Put("What is ---->"); Put(What,7); Put("<---"); New_Line;
   Put("Who is ----->"); Put(Who,8);  Put("<---"); New_Line;
   Put("TRUE is ---->"); Put(TRUE);   Put("<---"); New_Line;
   Put("FALSE is --->"); Put(FALSE);  Put("<---"); New_Line(2);

                                       -- Enumeration outputs
   Put("Today is --->"); Put(Today);   Put("<---"); New_Line;
   Put("Today is --->"); Put(Today,6); Put("<---"); New_Line;
   Put("Today is --->"); Put(Today,7); Put("<---"); New_Line;
   Put("WED is ----->"); Put(WED);     Put("<---"); New_Line;
   Put("WED is ----->"); Put(WED,5);   Put("<---"); New_Line(2);

end Formats;




-- Result of execution

-- Index is --->    27<---
-- Index is ---> 27<---
-- Index is --->      27<---
--
-- Count is --->    -7<---
-- Count is ---> -7<---
-- Count is --->      -7<---
--
-- Put(X_Value) --------> 3.14000E+00
-- Put(X_Value,5) ------>    3.14000E+00
-- Put(X_Value,5,5) ---->    3.14000E+00
-- Put(X_Value,5,5,0) -->    3.14000
--
-- Put(Size) --------> 24.33
-- Put(Size,5) ------>   24.33
-- Put(Size,5,5) ---->   24.33008
-- Put(Size,5,5,0) -->   24.33008
--
-- What is ---->TRUE<---
-- Who is ----->FALSE<---
-- What is ---->TRUE   <---
-- Who is ----->FALSE   <---
-- TRUE is ---->TRUE<---
-- FALSE is --->FALSE<---
--
-- Today is --->TUE<---
-- Today is --->TUE   <---
-- Today is --->TUE    <---
-- WED is ----->WED<---
-- WED is ----->WED  <---
```

If you examine the program e_c14_p1.ada, you will find that it does very little other than illustrate the various methods of outputting formatted data to the monitor. Several types and variables are declared, then six separate instantiations of **Ada.Text_IO** are declared for use with the six types used in the program. Actually, there is nothing here that is new to you, since we have covered all of these statements before, but they are included here in order to present an example of all of them in one place.

A few comments should be made at this point. Note the six instantiations of **Ada.Text_IO** packages given in lines 19 through 31. These are necessary in order to output the various types we are using in this program. The package instantiations must be declared after the various types are declared.

One of the instantiations could be eliminated by transforming the type of the **MY_INTEGER** type variable to type **INTEGER** prior to outputting it, but this is done for illustration. When we instantiate a package, we are actually using a copy of a generic package, which will be the subject of a detailed study in part 2 of this tutorial.

After you have studied the program and understand it, compile and run it for an elaborate set of formatted output examples. Your compiler may output the float and fixed outputs slightly differently than that given in the result of execution due to different defaults assigned by your compiler.

FORMATTED OUTPUT TO A FILE

Example program ------> e_c14_p2.ada

```
                                       -- Chapter 14 - Program 2
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure EasyOut is

   Turkey : FILE_TYPE;

begin
                             -- First we create the file
   Create(Turkey, Out_File, "TEST.TXT");
                             -- Then we write to it
   Put_Line(Turkey, "This is a test of turkey");
   Put(Turkey, "and it should work well.");
   New_Line(Turkey, 2);
   Put_Line("Half of the turkey test");

   Set_Output(Turkey);       -- Make Turkey the default output
      Put_Line("This is another test of turkey");
      Put("and it should work well.");
      New_Line(2);
      Put_Line(Standard_Output, "Half of the turkey test");
   Set_Output(Standard_Output); -- Return to the Standard default

   Put_Line("Back to the standard default output.");
   Close(Turkey);

end EasyOut;




-- Result of execution

-- (Output to the monitor follows)
--
-- Half of the turkey test
-- Half of the turkey test
-- Back to the standard default output

-- (Output to the file TEST.TXT)
--
-- This is a test of turkey
-- and it should work well.
--
-- This is another test of turkey
-- and it should work well.
```

The example file named e_c14_p2.ada contains examples of how to use a file in a very easy fashion. Since there is no standard method which is used by all operating systems to name files, we cannot depend on what the rules will be in order to use Ada on any system, so we need two file names in order to write to or read from a file. We need an internal name, which will be used by our program to refer to the file, and which follows the naming rules defined by Ada. We also need an external name which will be used by the operating system to refer to the file and whose name follows the rules dictated by the operating system. In addition to having the two names, we need a way to associate the names with each other, and tell the system that we wish to use the file so named. We do all of this with the **Create** procedure as shown in line 11 of this program.

The variable **Turkey**, is the internal file name which we declared in line 7 to be of type **FILE_TYPE**, a type exported by the package **Ada.Text_IO** for just this purpose. The "with Ada.Text_IO;" statement in line 2 makes the type **FILE_TYPE** available in this program. The last argument, which is either a string constant, as it is in this case, or a string variable, gives the external name of the file we wish to create and write to. Note carefully, that if you have a file named TEST.TXT in your default directory, it will be erased when this program is executed, because this statement will erase it. The second parameter in this procedure call, is either **In_File** if you intend to read from the file, or **Out_File** if you plan to write to it. There is no mode using text I/O in which you can both read from and write to the same file. In this case, we wish to write to the file, so we use the mode **Out_File** and create the file.

WRITING TO THE FILE

Line 13 looks rather familiar to us, since it is our old friend **Put_Line** with an extra parameter prior to the text we wish to output. Since **Turkey** is of type **FILE_TYPE**, the system is smart enough to know that this string will be output to the file with the internal name of **Turkey**, and the external name of TEST.TXT, so the string will be directed there along with the "carriage return/line feed". The next line will also be directed there, as will the 2 **New_Lines** requested in line 15. Line 16 does not have the filename of **Turkey** as a parameter, so it will be directed to the default output device, the monitor, in the same manner that it has been for all of this tutorial.

REDIRECTING THE OUTPUT

Line 18 contains a call to the procedure **Set_Output** which will make the filename which is given as the actual parameter the default filename. As long as this is in effect, any output without a filename will be directed to the file with the internal filename **Turkey**. In order to output some data to the monitor, it can still be done, but it must be directed there by using its internal filename as a parameter, the internal filename being **Standard_Output**. The group of statements given in lines 19 through 22 do essentially the same thing as those in lines 13 through 16. The default is returned to the standard output device in line 23, and the program completes normally.

CLOSING THE FILE

The simple statement given in line 26 is used to close the file when we are finished with it. The system only needs to be given the internal name of the file, since it knows the corresponding external filename. Failure to close the file will do no harm in such a simple program, because the operating system will close it automatically, but in a large program, it would be wise to close a file when it is no longer needed. There is probably an upper limit on how many files can be open at one time, and there is no sense wasting a file allocation on an unneeded file.

Be sure to compile and run this program, then check your default directory to see that you have the file named TEST.TXT in it containing the expected text.

MULTIPLE FILES OPENED AT ONE TIME

Example program ------> **e_c14_p3.ada**

```
                                                -- Chapter 14 - Program 3
with Ada.Text_IO, Ada.Integer_Text_IO;
```

```
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure MultOut is

   Numbers, Some_Text, More_Text : FILE_TYPE;
   Index : INTEGER;

begin

   Create(Numbers, Out_File, "INTDATA.TXT");
   Create(Some_Text, Out_File, "TEST.TXT");
   Create(More_Text, Out_File, "CHARACTS.TXT");

   for Count in 3..5 loop
      Put(Some_Text, "This line goes to the TEST.TXT file.");
      Put(More_Text, "This line goes to the CHARACTS.TXT file.");
      for Count2 in 12..16 loop
         Put(Numbers, Count + Count2);
      end loop;
      New_Line(More_Text);
      New_Line(Numbers, 2);
      New_Line(Some_Text);
   end loop;

   Put_Line("INTDATA.TXT, TEST.TXT, and CHARACTS.TXT are ready.");

   Close(Numbers);
   Close(Some_Text);
   Close(More_Text);

end MultOut;




-- Results of execution


-- (Output to the monitor)
--
-- INTDATA.TXT, TEST.TXT, and CHARACTS.TXT are ready.


-- (Output to the file named INTDATA.TXT)
--
--      15    16    17    18    19
--
--      16    17    18    19    20
--
--      17    18    19    20    21
--


-- (Output to the file named TEST.TXT)
--
-- This line goes to the TEST.TXT file.
-- This line goes to the TEST.TXT file.
-- This line goes to the TEST.TXT file.


-- Output to the file named CHARACTS.TXT)
--
```

```
-- This line goes to the CHARACTS.TXT file.
-- This line goes to the CHARACTS.TXT file.
-- This line goes to the CHARACTS.TXT file.
```

Examine the file e_c14_p3.ada for an example program in which we open and use 3 different files plus the monitor. The three internal filenames are declared in line 7, and all three are opened in the output mode with three different external names as shown in lines 12, 13, and 14.

We execute the loop in lines 16 through 25, where we write nonsense data to the three files, output a test string to the display in line 27, and finally close all three files. Note that we could have defined one of the files to be the default file and written to it without the filename, then used the filename for the standard output in line 27, achieving the same result.

The program is fairly simple, so after you feel you understand it, compile and run it. After a successful run, examine the default directory to see that the three files exist and contain the expected results. Do not erase the generated files, because we will use them as example input files in the next few programs.

INPUTTING CHARACTERS FROM A FILE

Example program ------> **e_c14_p4.ada**

```
                                       -- Chapter 14 - Program 4
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure CharIn is

   My_File  : FILE_TYPE;
   One_Char : CHARACTER;

begin

   open(My_File, In_File, "CHARACTS.TXT");

   loop       -- Read one character at a time and display it
      exit when End_Of_File(My_File);
      Get(My_File, One_Char);
      Put(One_Char);
   end loop;
   New_Line(2);

   Reset(My_File);  -- Reset and start over with the same file

   loop       -- Read and display but search for End of lines
      exit when End_Of_File(My_File);
      Get(My_File, One_Char);
      if End_Of_Line(My_File) then
         Put("<--- End of line found");
         New_Line;
      else
         Put(One_Char);
      end if;
   end loop;
   New_Line;

   Reset(My_File);  -- Reset and start over the third time

              -- Read and display but search for End of lines
   loop       -- using a look ahead method
      exit when End_Of_File(My_File);
      Get(My_File, One_Char);
      Put(One_Char);
      if End_Of_Line(My_File) then
         Put("<--- End of line found");
         New_Line;
      end if;
   end loop;

   Close(My_File);

end CharIn;




-- Result of execution

-- (Note; the first line is a full 80 columns wide.)
-- This line goes to the CHARACTS.TXT file.This line goes to...
-- This line goes to the CHARACTS.TXT file.
--
-- This line goes to the CHARACTS.TXT file<--- End of line found
-- This line goes to the CHARACTS.TXT file<--- End of line found
-- This line goes to the CHARACTS.TXT file<--- End of line found
--
-- This line goes to the CHARACTS.TXT file.<--- End of line found
-- This line goes to the CHARACTS.TXT file.<--- End of line found
-- This line goes to the CHARACTS.TXT file.<--- End of line found
```

Examine the file e_c14_p4.ada for an example of how to read **CHARACTER** type data from a file. In this example the file name **My_File** is declared as a **FILE_TYPE** variable, then used to open CHARACTS.TXT for reading since the **In_File** mode is used. In this case, the file must exist, or an exception will be raised. The file pointer is set to the beginning of the file by the **Open** procedure so we will read the first character in the file the first time we read from the file.

The loop in lines 14 through 18 causes us to read a character from the file and display it on the monitor each time we pass through the loop. The function **End_Of_File** returns a **TRUE** when the next character to be read is the end of file character for your particular implementation. We do not need to know what the end of file character is, it is up to the compiler writer to find out what it is and return a value of **TRUE** when the system finds it. The entire file is therefore displayed on the monitor by this program, but with one slight deviation from what you would expect. Because of the way Ada is defined, the end of line characters are simply ignored by the **Get** procedure, and since they are not read in, they cannot be output to the monitor either. The end result is that the characters are all output in one long string with no line feeds. You will see this later when you compile and run the program.

We will continue our study in this program and see a much better way to read and display the data.

THE RESET PROCEDURE

The **Reset** procedure in line 21 resets the file named as a parameter, which simply means that the file pointer is moved to the first character once again. The file is ready to be read in another time. The loop in lines 23 through 32 is the same as the previous loop except for the addition of another new function. When the end of a line is found, the function **End_Of_Line** returns a **TRUE**, and the program displays the special little message in line 27. In addition to the message, the **New_Line** procedure is called to supply the otherwise missing end of line.

Unfortunately, we still have a problem because the **End_Of_Line** is actually a lookahead function and becomes **TRUE** when the next character in the buffer is an end of line character. When the program is executed, we find that it does not output the last character of each line in this loop, because we never execute the **else** clause in the **if** statement for that character. When the program is executed, you will notice that the periods at the end of the sentences are not displayed. The next loop in this program will illustrate how to input and output the data correctly considering all of the Ada idiosyncrasies.

If you recall, we stated at the beginning of this lesson that input and output programming in Ada was at best a compromise. Don't worry, you will be able to input or output anything you wish to very soon.

A CORRECT CHARACTER INPUT/OUTPUT LOOP

The file is once again reset in line 35, and a third loop in lines 38 through 46 reads and displays the file on the monitor. This time when we go through the loop, we recognize that the end of line is reading one character ahead in the input file, not the one we just read, and we adjust the program accordingly. We output the character, then check for end of line, and call the **New_Line** procedure if we detect an end of line. You will see, when you run it, that the last character on the line will be displayed as desired.

It should be pointed out to you that the **End_Of_File** is another lookahead function and must be tested after we output the last character read in. This loop does just that, because the end of file is acted on after the character is output.

Be sure to compile and run this program. It would be interesting to edit the input file, CHARACTS.TXT, adding a few characters, then rerunning the program to see how it handles the new data.

STRING INPUT AND OUTPUT

Example program ------> e_c14_p5.ada

```
                                       -- Chapter 14 - Program 5
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure StringIn is

   My_File   : FILE_TYPE;
   My_String : STRING(1..10);

begin

   Open(My_File, In_File, "CHARACTS.TXT");

   loop       -- Read one string at a time and display it
      exit when End_Of_File(My_File);
      Get(My_File, My_String);
      Put(My_String);
   end loop;
   New_Line(2);

   Reset(My_File);  -- Reset and start over with the same file

   loop       -- Read and display but search for End of lines
      exit when End_Of_File(My_File);
      Get(My_File, My_String);
      Put(My_String);
      if End_Of_Line(My_File) then
         Put_Line("<--- End of line found");
      else
         Put_Line("<--- 10 characters");
      end if;
   end loop;
   New_Line;

   Close(My_File);

end StringIn;




-- Result of execution

-- (Note; The first line is a full 80 columns wide.)
-- This line goes to the CHARACTS.TXT file.This line goes to ...
-- This line goes to the CHARACTS.TXT file.
--
-- This line <--- 10 characters
-- goes to th<--- 10 characters
-- e CHARACTS<--- 10 characters
-- .TXT file.<--- End of line found
-- This line <--- 10 characters
-- goes to th<--- 10 characters
-- e CHARACTS<--- 10 characters
-- .TXT file.<--- End of line found
-- This line <--- 10 characters
-- goes to th<--- 10 characters
-- e CHARACTS<--- 10 characters
-- .TXT file.<--- End of line found
```

The next example program, e_c14_p5.ada, illustrates how to read a string from an input file. In much the same manner as for the character, the end of line character is ignored when encountered in the input file. The result is that when the first loop is run, no "carriage returns" are issued and the text is all run together. The second loop, however, uses the lookahead method and calls the **New_Line** procedure when it detects the end of line in the input stream.

The big difference in this program and the last is the fact that a **STRING** type variable is used for input here and a **CHARACTER** type variable was used in the last one.

I PLAYED A TRICK ON YOU

This program has a bit of a trick in it. The input file contains an exact multiple of 10 characters in each line, and the input variable, of type **STRING**, has ten characters in it. There is therefore no real problem in reading into the ten character string, but if the lines were not exact multiples of ten characters, we would have gotten an input error. If you remember how difficult the **STRING** variable was to work with when we studied strings, you will understand that we have the same problem here. When we get to chapter 16 with its example programs, we will see how we can greatly improve the string capability with an add-on library of dynamic string routines. Also, Ada 95 has several dynamic string packages available for your use.

Be sure to compile and run this program, then add a few characters to some of the lines in the file named CHARACTS.TXT to see the result of trying to read in an odd group of extra characters.

NOW TO READ IN SOME NUMBERS

Example program ------> **e_c14_p6.ada**

```
                                       -- Chapter 14 - Program 6
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;
```

```
procedure IntIn is

   Integer_File : FILE_TYPE;
   Index        : INTEGER;

begin

   Open(Integer_File, In_File, "INTDATA.TXT");

   while not End_Of_File(Integer_File) loop
      if End_Of_Line(Integer_File) then
         New_Line;
         Skip_Line(Integer_File);
      else
         Get(Integer_File, Index);
         Put("The value read in is");
         Put(Index, 6);
         New_Line;
      end if;
   end loop;

   Close(Integer_File);

end IntIn;




-- Result of execution

-- The value read in is    15
-- The value read in is    16
-- The value read in is    17
-- The value read in is    18
-- The value read in is    19
--
-- The value read in is    16
-- The value read in is    17
-- The value read in is    18
-- The value read in is    19
-- The value read in is    20
--
-- The value read in is    17
-- The value read in is    18
-- The value read in is    19
-- The value read in is    20
-- The value read in is    21
```

Computers are very good at working with numbers, and in order to work with them, we must have a way to get them into the computer. The example file e_c14_p6.ada illustrates how to read **INTEGER** type data into the computer from a file. Much like the last two programs, we open a file for reading, and begin executing a loop where we read an **INTEGER** type number as input each time through the loop. We are reading an **INTEGER** type number because that is the type of **Index**, the second actual parameter in the **Get** procedure call. Since it is an **INTEGER** type number, the system will begin scanning until it finds a valid **INTEGER** type number terminated by a space. It will then return that number, in the computer's internal format, assigned to the variable we supplied to it, and make it available for our use like any other **INTEGER** type variable. If any data other than **INTEGER** type data were to be encountered in the input file before it found a valid

INTEGER value, an exception would be raised, and the program would be terminated.

The loop continues in much the same manner as the last two programs until it detects the end of file, at which time it stops. There is one other slight difference when using numeric inputs, the system does not read over the end of line character, because it doesn't know how to get across it. It gets stuck trying to read the next data point but never gets to it because the end of line is blocking it. When the end of line is detected, or at any other time, you can issue the function **Skip_Line** which tells it to go to the beginning of the next line for its next input data. At any time then, you can begin reading on the next line for your next data point.

Be sure to compile and run this program, then change the spacing of some of the numbers in the input file to see that they are read in with a free form spacing.

WHAT ABOUT FLOATING POINT DATA INPUTS?

Once you understand the method of reading **INTEGER** type data in, you can read any other types in by using the same methods. Just remember that the data you read in must be of the same type as the variable into which it is being read and you will have little difficulty.

WHAT ABOUT READING FROM THE KEYBOARD?

Reading from the keyboard is similar to reading from a file, except that it uses the predefined internal filename, **Standard_Input**, and if you omit a filename reference, the input will come from the keyboard. It should be pointed out that the operating system will probably buffer the input data automatically and give it to your program as a complete string when you hit the return key. It will allow you to enter a complete string of as many integer values as you desire, and when you hit the return, the entire string will be input and processed.

Modify the last file to read from the keyboard, and enter some numbers after compiling and running it. Note that there is no **End_Of_File** when reading from the keyboard. Use a loop with a fixed number of passes, or use an infinite loop and quit when a certain number is input such as -1.

HOW DO WE PRINT DATA ON THE PRINTER?

Example program ------> e_c14_p7.ada

```
                                       -- Chapter 14 - Program 7
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure PrintOut is

   Turkey : FILE_TYPE;

begin
                             -- First we create the file
   Create(Turkey, Out_File, "LPT1");
                             -- Then we write to it
   Put(Turkey, "This is a test of turkey");
   New_Line(Turkey);
   Put(Turkey, "and it should work well.");
   New_Line(Turkey, 2);
   Put("Half of the turkey test.");
   New_Line;

   Set_Output(Turkey);       -- Make Turkey the default output
      Put("This is another test of turkey");
      New_Line;
      Put("and it should work well.");
      New_Line(2);
      Put(Standard_Output, "Half of the turkey test.");
      New_Line(Standard_Output);
   Set_Output(Standard_Output); -- Return to the Standard default

   Put("Back to the standard default output.");

   Close(Turkey);

end PrintOut;




-- Results of execution

-- (Output to the monitor)
--
-- Half of the turkey test.
-- Half of the turkey test.
-- Back to the standard default output.

-- (Output to the printer)
--
-- This is a test of turkey
-- and it should work well.
--
-- This is another test of turkey
-- and it should work well.
--
```

The program named e_c14_p7.ada illustrates how to send data to your printer. The only difference in sending data to the printer, from sending data to a disk file, is the use of the predefined name **LPT1** for the external file name. This file is defined as the printer rather than a file, and as you can see from the program, it operates no differently than any other file.

Other names may be applicable for the printer also, such as **PRN**, **COM1**, etc. Check your documentation for the predefined names that mimic a file.

Be sure your printer is on line, then compile and run this program.

TEXT IO PRAGMAS

A pragma is a suggestion to the compiler and can be ignored by any implementation according to the definition of Ada. Refer to Annex L of the Ada 95 Reference Manual (ARM) for the definition of the predefined pragmas **LIST** and **PAGE**. Since these may not be available with any particular compiler, their use should be discouraged in order to attain a high degree of program portability.

PAGE AND LINE CONTROL SUBPROGRAMS

Refer to the definition of **Ada.Text_IO** in Annex A.10.1 of the ARM for a large assortment of page and line control subprograms for use with text I/O. These are required to be available with every compiler, so you should study them to gain an insight into the page and line control subprograms available to you in any Ada program. There should not be anything too difficult for you to understand in these definitions.

PROGRAMMING EXERCISES

1. Modify e_c14_p4.ada to read from the keyboard and echo data to the monitor until a Q is entered.(Solution)

```
                          -- Chapter 14 - Programming exercise 1
with Ada.Text_IO;
use Ada.Text_IO;
```

```
procedure CH14_1 is

   One_Char : CHARACTER;

begin

   Put_Line("Input characters to display, enter Q to stop.");

   loop       -- Read one character at a time and display it
      Get(One_Char);
      Put(One_Char);
      New_Line;
      exit when One_Char = 'Q';
   end loop;
   New_Line(2);

end CH14_1;




-- Result of execution

-- (The output depends on the input.)
```

2. Write a program named FLOATIN.ADA that reads floating point data from a file and displays it. There is no standard on floating point input. Some compilers require a decimal point with at least one digit before and after the decimal point. Some may not even require a decimal point but will supply it to an input that looks like an **INTEGER** value. Experiment and determine the characteristics of your compiler.(Solution)

```
                         -- Chapter 14 - Programming exercise 2
with Ada.Text_IO, Ada.Float_Text_IO;
use Ada.Text_IO, Ada.Float_Text_IO;

procedure Ch14_2 is

   Float_File : FILE_TYPE;
   Float_Dat  : FLOAT;

begin

   Open(Float_File, In_File, "FLTDATA.TXT");

   while not End_Of_File(Float_File) loop
      if End_Of_Line(Float_File) then
         New_Line;
         Skip_Line(Float_File);
      else
         Get(Float_File, Float_Dat);
         Put("The value read in is");
         Put(Float_Dat);
         New_Line;
      end if;
   end loop;

   Close(Float_File);

end Ch14_2;
```

```
-- Result of execution

-- (The output depemds on the input values.)
```

Ada Tutorial - Chapter 15

PACKAGES

PACKAGES ARE WHY ADA EXISTS

One of the biggest advantages of Ada, over most other programming languages, is its well defined system of modularization and separate compilation. Even though Ada allows separate compilation, it maintains the strong type checking among the various compilations by enforcing rules of compilation order and compatibility checking. Ada uses separate compilation, but FORTRAN, as a classic example, uses independent compilation, in which the various parts are compiled with no knowledge of the other compilation units with which they will be combined. As we progress through this material, and additional material to come later, do not be discouraged if you find many things to keep in mind when doing separate compilations. The rules are not meant to be roadblocks, but are actually benefits for you when you are working on a large complex system.

LET'S LOOK AT A PACKAGE

Example program ------> e_c15_p1.ada

```
                                       -- Chapter 15 - Program 1

                 -- Interface of AdderPkg
package AdderPkg is
   type MY_ARRAY is array(INTEGER range <>) of FLOAT;
   procedure Add_Em_Up(In_Dat : in     MY_ARRAY;
                       Sum    :    out FLOAT);
end AdderPkg;


                 -- Implementation of AdderPkg
package body AdderPkg is
   procedure Add_Em_Up(In_Dat : in     MY_ARRAY;
                       Sum    :    out FLOAT) is
   Total : FLOAT;
   begin
      Total := 0.0;
      for Index in In_Dat'FIRST..In_Dat'LAST loop
         Total := Total + In_Dat(Index);
      end loop;
      Sum := Total;
   end Add_Em_Up;
end AdderPkg;




-- Result of execution

-- (This is not a stand alone package, so it has no output.)
```

Examine the file named e_c15_p1.ada for our first example of a separately compiled Ada package. A package, as the term is used in Ada, refers to a collection of related entities, the collection being composed of procedures, functions, variables, constants, types, subtypes, and even other packages. In our present example, the package is composed of one type, and one procedure.

THE SPECIFICATION OF THE PACKAGE

Lines 4 through 8 define the specification of the package named **AdderPkg**, which is actually a very simple package, and purposely kept simple for illustrative purposes. The type **MY_ARRAY** is defined, as well as the procedure heading for **Add_Em_Up** in the specification part of the package. The only things a user needs to know about the package in order to use it are defined in the package specification, so it becomes the interface to the outside world. With a few defining statements, this is all the user would need to know about the package, and he could be isolated from the actual details of how the procedure does its job. We will see more about the topic of information hiding later in this chapter.

Note that any entity that is declared in the specification part of the package can be used in any other package that **with**'s this package.

We have an unconstrained array type declared in line 5 which we have not yet studied in this tutorial. The range for the subscript is defined by the "<>", which is a box that must be filled in later when we define the actual type. We will cover this in detail in the chapter on advanced array topics.

THE BODY OF THE PACKAGE

Lines 12 through 23 define the body of the package, and it is distinguished from the specification by the reserved word **body** in its header, and by the fact that the procedure is completely defined here. Anything defined or declared in the specification part of the package is available for use here, just as if it were defined at the beginning of this section. The procedure header is redefined here in full, and it must exactly match the definition in the specification or you will get a compile error and no compilation. There is nothing different about this procedure from any other procedure we have seen, it just happens to be in the package body.

Note that any types, variables, constants, procedures, or functions, can be declared in the package body for use within the body, but none of them are available outside of the package because they are not defined in the specification part of the package. The variable declared in line 15 named **Total** is not available outside of this package and there is no way it can be referred to outside of the package without being declared in the package specification. In fact, since it is embedded within the procedure, it would be hidden anyway, but the fact remains that no entities are available outside of the package except those that are declared in the specification of the package.

It is not legal to place any subprogram bodies in the package specification.

Note that since the array has no defined limits, we must use attributes to define the loop range. Even though you may find this a bit confusing, you will appreciate the flexibility found here after we study some of the more advanced topics.

IT CAN BE COMPILED BUT NOT EXECUTED

This file can be compiled, but it cannot be linked and executed because it is not a complete program, it is only a package containing a type and a procedure which can be called from another program just like we have been calling the procedure **Put** in the **Ada.Text_IO** package throughout this tutorial.

One other point must be made before we look at a program to use this package, the specification and the body do not need to be in the same file, they can be contained in separate files and compiled separately. If this is the case, the specification must be compiled prior to compiling the body because the body uses information generated during compilation of the specification. Actually, even though they are in one file in this example, they are considered separately by the compiler, being compiled in serial fashion. This file is said to be composed of two compilation units.

FILENAME VERSUS PACKAGE NAME

In all of the example programs so far in this tutorial, we have used the same name for the filename and the program name, or the procedure name. This is not really necessary, but it was felt that an additional level of complexity should be delayed until later. We have arrived at the time when an explanation is needed.

When Ada compiles a program, it adds the result of the compilation into its library using the program name, not the filename. This library entry includes all information needed to link the program with the other needed library entries. Later when you use the linker supplied with your compiler, the linker will collect all of the necessary compiled packages and subprograms and combine them into an executable program.

In order to compile a program, it is necessary to give the filename so the operating system can find the program for the Ada compiler, but in order to link a program, the filename is no longer of any use because the name that really matters is the program name, and the program name is what the linker uses. Ada allows identifiers to be of any arbitrary length, but your operating system has some length limit, eight if you are using MS_DOS, therefore the Ada compiler may need to somehow make up a new name if intermediate results are put into individual files. It will be up to you to determine how your compiler stores intermediate results and how it makes up filenames for these results.

For simplicity, therefore, all program names have been limited to eight characters, and the same name is used for the filename throughout most of this tutorial.

NOW TO USE THE NEW PACKAGE

Example program ------> **e_c15_p2.ada**

```
                                       -- Chapter 15 - Program 2
with Ada.Text_IO, Ada.Float_Text_IO;
use Ada.Text_IO, Ada.Float_Text_IO;

with AdderPkg;
use AdderPkg;

procedure Adder1 is

   FIRST : constant := 2;
   LAST  : constant := 7;
   Sum_Of_Values : FLOAT;

   New_Array : MY_ARRAY(FIRST..LAST);

   procedure Summer(In_Dat : MY_ARRAY;
                    Sum    : out FLOAT) renames AdderPkg.Add_Em_Up;
begin
   for Index in New_Array'FIRST..New_Array'LAST loop
      New_Array(Index) := FLOAT(Index);
   end loop;

   Put_Line("Call Add_Em_Up now");
   Add_Em_Up(New_Array, Sum_Of_Values);
   Put("Back from Add_Em_Up, total is");
   Put(Sum_Of_Values, 5, 2, 0);
   New_Line;

          -- The next three statements are identical
   Add_Em_Up(New_Array, Sum_Of_Values);
   AdderPkg.Add_Em_Up(New_Array, Sum_Of_Values);
   Summer(New_Array, Sum_Of_Values);

end Adder1;
```

The example file named e_c15_p2.ada, illustrates how to use the previously studied package. There is nothing different about this program from any other program we have used except that it uses the package we defined and named **AdderPkg**, which it acknowledges in lines 5 and 6 where it tells the system to **with** the package and to **use** it in the program at hand. It also uses the renaming statement in line 16 which we will discuss later. The remainder of the program is simple, and you should have no trouble deciphering it. The primary purpose of these two examples is to illustrate how to write a library package.

In line 14 we declare an array of type **MY_ARRAY** and at this time we supply the range limits for the subscript. You may begin to see the flexibility in this method of array declaration, but we will study it in detail later.

THE with CLAUSE

It is finally time for a complete definition of just what the **with** clause does for us. When we **with** a package into our program, we are telling the system that we wish to have everything that is declared in the specification of that package available for our use in this program. Accordingly, the system will look at the items declared in the specification for that package and every time we use one of those items, it will see that we have the correct number of parameters and that we have the types declared for each parameter correctly. Note that this happens during compilation and explains why Ada is said to be compiled separately, but not independently of other compilation units. In a sense, the resources available in the **with**ed package act as though they are extensions to the Ada programming language. Because of this, the **with** clause is called a context clause.

When you arrive at the linking operation, the **with**ed packages are automatically added into the executable file, as are any other packages that are **with**ed into your program. Note that the package named **Standard** is automatically **with**ed into every Ada program, subprogram, or package. The package named **Standard** defines many of the predefined entities such as **INTEGER**, **BOOLEAN**, **FLOAT**, etc.

In a large program, it is possible to **with** the same package several times since it is used in several packages. You can be assured that only one copy of the package will be included during the linking operation. Multiple copies will not be stored.

One other point must be made before leaving this topic, and that is the fact that all **with** clauses must be at the beginning of the program or package. There is a good reason for this. The dependent packages, and hence the overall program structure must be given at the beginning of each package making it easy to ascertain the overall structure without being forced to search through the entire listing for each package.

THE use CLAUSE

The **use** clause in Ada allows you to use a shorthand when naming procedures, functions, variables, etc, from a package. Instead of using the extended naming convention, or the so called "dot" notation, and including the package name followed by the procedure name, dotted together, you can simply use the procedure name and let the system figure out what package it is coming from. In most of the programs we have studied so far, we have included the **Ada.Text_IO** package in a **use** clause. If the **use** clause were omitted we would have to identify the package each time one of the procedures is used. Put("This is Ada"); would have to be changed to read Ada.Text_IO.Put("This is Ada");, which clutters up the listing a bit but removes all ambiguity.

Because it is possible to get a different procedure than the one you are expecting under very unusual conditions of overloaded procedure names, the use of the **use** clause is falling into some disrepute in the software engineering literature. Without the **use** clause, you are forced to type in additional information for each procedure call. The presence of the package name prepended to each subprogram call however, leads to no ambiguity and therefore follows the basic premise of Ada that a program is written once but read many times. The extra keystrokes are worth the trouble to include them.

Use of the **use** clause is a matter of personal taste or possibly a style dictated by a project style guide.

RENAMING A PROCEDURE

A procedure can be renamed in order to reduce the length of the identifier, especially if a rather long extended name is required. It may be better to use a more descriptive name based on the actual use of a general purpose procedure. The method of renaming is illustrated in lines 16 and 17 of this program. Of most importance is the fact that the entire list of formal parameters must be repeated. This is done so that the definition of the new procedure name is complete and should be of help during program debugging.

Use of the new name, which is of course only a synonym and not a new procedure, is illustrated in line 32 of this program.

If you compiled the previous file, named e_c15_p1.ada, you can compile, link, and execute this one to see that the system knows how to link the two together.

COMBINING FILES

If you wish, you could combine the two files, provided you appended the second file to the end of the first. The compiler would then compile all three in succession, after which you could link the results, and execute the result. The library files must be compiled first so that the compiler can check the types in the procedure call to see that they agree, so putting them before the calling program conforms to this rule. If you did put them in a single file, you would still need the statements in lines 5 and 6 of e_c15_p2.ada to tell the system to **with** and **use** the library file defined earlier in the file, because the compiler would consider them to be three separate compilations.

ANOTHER METHOD OF COMBINING FILES

Example program ------> **e_c15_p3.ada**

```
                                               -- Chapter 15 - Program 3
with Ada.Text_IO, Ada.Float_Text_IO;
use Ada.Text_IO, Ada.Float_Text_IO;

procedure Adder2 is

   FIRST : constant := 2;
   LAST  : constant := 7;
   Sum_Of_Values : FLOAT;

                      -- Interface of Package1
   package AdderPkg is
      type MY_ARRAY is array(INTEGER range <>) of FLOAT;
      procedure Add_Em_Up(In_Dat : in     MY_ARRAY;
                          Sum    :    out FLOAT);
   end AdderPkg;


   use Adder2.AdderPkg;
   New_Array : MY_ARRAY(FIRST..LAST);


                       -- Implementation of Package1
   package body AdderPkg is
      procedure Add_Em_Up(In_Dat : in     MY_ARRAY;
                          Sum    :    out FLOAT) is
      Total : FLOAT;
      begin
         Total := 0.0;
         for Index in In_Dat'FIRST..In_Dat'LAST loop
            Total := Total + In_Dat(Index);
         end loop;
         Sum := Total;
      end Add_Em_Up;
   end AdderPkg;


begin
   for Index in New_Array'FIRST..New_Array'LAST loop
      New_Array(Index) := FLOAT(Index);
   end loop;

   Put_Line("Call Add_Em_Up now");
   Add_Em_Up(New_Array, Sum_Of_Values);
   Put("Back from Add_Em_Up, total is");
   Put(Sum_Of_Values, 5, 2, 0);
   New_Line;

end Adder2;




-- Result of execution

-- Call Add_Em_Up now
-- Back from Add_Em_Up, total is   27.00
```

The example program named e_c15_p3.ada illustrates another way to combine the last two files, in this case including the package in the declaration part of the program. The specification part of the package is in lines 12 through 16, and the body is in lines 24 through 35. In this case, a new type is defined between the two parts to illustrate that it can be done. Since the package is compiled as a part of the main program, it does not have to be mentioned in a **with** statement. The compiler knows that it is a part of the program, but the **use** must be mentioned to tell the system where to get the procedure name and the type. Of course the **use** can be omitted and the extended naming convention used for all references to the package.

Even though the body is defined after the variable **New_Array** is declared in line 20, this variable is not directly available for use in the body, because the package structure effectively builds a strong wall around the enclosed statements, and nothing can get in or out. The only inputs to and outputs from the body are those defined in the specification part of the package. Of course the variable **New_Array** is available to the procedure because it is passed in as a parameter, but is not directly visible.

One other difference occurs here that is different from the last two files. This embedded package is not available for use by any other program, because it is enclosed within this program, and is not therefore a library package. The entire file contains only one compilation unit.

ANOTHER KIND OF SEPARATE COMPILATION

Example program ------> **e_c15_p4.ada**

```
                                       -- Chapter 15 - Program 4
with Ada.Text_IO, Ada.Float_Text_IO;
use Ada.Text_IO, Ada.Float_Text_IO;

procedure Adder3 is

   FIRST : constant := 2;
   LAST  : constant := 7;
   Sum_Of_Values : FLOAT;

                      -- Interface of AdderStb
   package AdderStb is
      type MY_ARRAY is array(INTEGER range <>) of FLOAT;
      procedure Add_Em_Up(In_Dat : in     MY_ARRAY;
                          Sum    :    out FLOAT);
   end AdderStb;


   use AdderStb;
   New_Array : MY_ARRAY(FIRST..LAST);


                       -- Implementation of AdderStb
   package body AdderStb is separate;


begin
   for Index in New_Array'FIRST..New_Array'LAST loop
      New_Array(Index) := FLOAT(Index);
   end loop;

   Put_Line("Call Add_Em_Up now.");
   Add_Em_Up(New_Array, Sum_Of_Values);
   Put("Back from Add_Em_Up, total is");
   Put(Sum_Of_Values, 5, 2, 0);
   New_Line;

end Adder3;




-- Result of execution

-- Call Add_Em_Up now
-- Back from Add_Em_Up, total is   27.00
```

The example file named e_c15_p4.ada illustrates still another method of separate compilation. This program is identical to the last except that the package body is removed to a separate file for separate compilation, and is called a stub. The statement in line 24 indicates to the compiler that the body will be found elsewhere. Although this illustrates separate compilation of a package body, the same method can be used for separate compilation of a procedure. This could be used if you wished to remove a large procedure from the logic defined here to make it more manageable.

Example program ------> **e_c15_p5.ada**

```
                                       -- Chapter 15 - Program 5
separate(Adder3)
                 -- Implementation of AdderStb
package body AdderStb is
   procedure Add_Em_Up(In_Dat : in     MY_ARRAY;
                       Sum    :    out FLOAT) is
   Total : FLOAT;
   begin
      Total := 0.0;
      for Index in In_Dat'FIRST..In_Dat'LAST loop
         Total := Total + In_Dat(Index);
      end loop;
      Sum := Total;
   end Add_Em_Up;
end AdderStb;
```

The separately compiled body is found in the file named e_c15_p5.ada, which begins with the reserved word **separate** which indicates that this is a stub. The main program, or whatever other package, procedure, or function, that uses this stub is defined in parentheses following the reserved word **separate** to tell the compiler where this is used. This stub cannot be called or used by any other program, because it is in truth part of the program **Adder3**, not a general purpose program. Any variables, types, procedures, etc, that are available for use at the point where the stub is used, are available at the point where the stub is defined. The stub therefore has the same environment as the environment existing at the point of use, in this case line 24 of e_c15_p4.ada.

ORDER OF COMPILATION FOR THE STUB

Since all of the variables, types, etc must be made available to the stub, the using program must be compiled before the stub itself is compiled. After both are compiled, the program can be linked and executed. You should compile e_c15_p4.ada first, then e_c15_p5.ada should be compiled, and finally ADDER3 should be linked. You will then have an executable ADDER3 program.

ORDER OF COMPILATION IS NOT MYSTERIOUS

The example files included with this chapter are intended to illustrate to you the required order of compilation in a meaningful way, not as a group of rules to be memorized. If you understand the dependencies of files on one another, the order of compilation will make sense, and you will be able to intelligently arrange your programs to use the Ada type checking between the various separately compiled files. Remember that Ada, unlike Pascal, was designed to allow the development of huge programs that require separate compilation facilities, and yet retain the strong type checking between modules.

A PACKAGE WITH AN INITIALIZATION PART

Example program ------> **e_c15_p6.ada**

```
                                       -- Chapter 15 - Program 6

                 -- Interface of AdderPkg
package AdderPkg is
   Total : FLOAT;               -- Global variable
   type MY_ARRAY is array(INTEGER range <>) of FLOAT;
   procedure Add_Em_Up(In_Dat : in     MY_ARRAY;
                       Sum    :    out FLOAT);
end AdderPkg;


                 -- Implementation of AdderPkg
package body AdderPkg is
   procedure Add_Em_Up(In_Dat : in     MY_ARRAY;
                       Sum    :    out FLOAT) is
   begin
      for Index in In_Dat'FIRST..In_Dat'LAST loop
         Total := Total + In_Dat(Index);
      end loop;
      Sum := Total;
   end Add_Em_Up;

begin            -- Initialization section
   Total := 0.0;
end AdderPkg;
```

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

with AdderPkg;
use AdderPkg;

procedure Adder4 is

   package Flt_IO is new Ada.Text_IO.Float_IO(FLOAT);
   use Flt_IO;

   FIRST : constant := 2;
   LAST  : constant := 7;
   Sum_Of_Values : FLOAT;

   New_Array : MY_ARRAY(FIRST..LAST);

   procedure Summer(In_Dat : MY_ARRAY;
                    Sum    : out FLOAT) renames AdderPkg.Add_Em_Up;
begin
   for Index in New_Array'FIRST..New_Array'LAST loop
      New_Array(Index) := FLOAT(Index);
   end loop;

   Put_Line("Call Add_Em_Up now.");
   Add_Em_Up(New_Array, Sum_Of_Values);
   Put("Back from Add_Em_Up, total is");
   Put(Sum_Of_Values, 5, 2, 0);
   New_Line;

          -- The next three statements are identical
   Add_Em_Up(New_Array,Sum_Of_Values);
   AdderPkg.Add_Em_Up(New_Array, Sum_Of_Values);
   Summer(New_Array, Sum_Of_Values);

end Adder4;




-- Result of execution

-- Call Add_Em_Up now
-- Back from Add_Em_Up, total is   27.00
```

The example file named e_c15_p6.ada illustrates one more feature of a package. This program is nearly identical to the first program in this chapter. Both files which comprise the first example program have been incorporated into a single file for ease of compilation, and the variable named **Total** has been made global within the package. Since it is global within the package, it can be referred to by any subprogram in the package or in the initialization section as illustrated in lines 23 and 24. This section is optional but can be included in any package.

Note that the initialization section is only executed once, at load time. For that reason, this program will not operate exactly as the first one did. The result that is output is identical in both cases, but the additional calls will not produce the same result because the variable **Total** is not cleared to zero during each call in this example program as it is in the first example. The initialization section only initializes the variable once, and it is impossible to call this section of code from within the executable part of the package.

A FEW OF THE STANDARD PACKAGES

Ada 95 provides many new packages to improve the programming model and prevents errors on the part of the programmer, and to provide additional conveniences when programming. We have already discussed a few of the additional packages during our study of Ada earlier in this tutorial. We will make a few comments on a few more that you may find useful, but really didn't fit in well anywhere yet in the tutorial.

package **Standard** - This is the root of all packages in Ada 95 and is where such things as **INTEGER**, **FLOAT**, **BOOLEAN**, and other such entities are defined. It is inherently **with**ed into every Ada compilation unit automatically.

package **Ada.Command_Line** - This is actually a part of the above mentioned **Standard** package but that fact is really invisible to you. This package provides the ability to retrieve any parameters provided by the user when he begins execution of the resulting program. This package provides a means to retrieve those parameters in a portable manner. This did not exist in Ada 83, so each compiler writer came up with a non-portable way to do this.

package **Ada.Numerics.Elementary_Functions** - This package provides the trig functions along with a floating point exponentiation operator, and many other math functions.

package **Ada.Characters.Latin_1**- This package replaces the **ASCII** package which was available with Ada 83 but is now obsolete. This new package provides definitions of the ASCII character set.

INTERFACES TO OTHER LANGUAGES

Annex B of the ARM defines the method of interfacing to other languages which your compiler may or may not support. The packages named **Interface.C**, **Interface.COBOL**, and **Interface.Fortran**, if available with your compiler, will provide you with the ability to combine code in those languages with your Ada code in a well defined manner. Of special importance is the method of handling strings, pointers, arrays, and structures since they can be so different for other languages. The interfaces consist of various pragmas and subprograms to define the interface characteristics. You can study the details on your own, if you ever have the need to interface to a language supported by your Ada compiler.

PROGRAMMING EXERCISES

1. Remove the package body from e_c15_p3.ada and make it a stub. Compile the two resulting files in the correct order and link and execute the resulting program.(Solution 1)(Solution 2)

```
                          -- Chapter 15 - Programming example 1
with Ada.Text_IO, Ada.Float_Text_IO;
use Ada.Text_IO, Ada.Float_Text_IO;

procedure CH15_1A is

   FIRST : constant := 2;
   LAST  : constant := 7;
   Sum_Of_Values : FLOAT;

                      -- Interface of Package1
   package AdderPkg is
      type MY_ARRAY is array(INTEGER range <>) of FLOAT;
      procedure Add_Em_Up(In_Dat : in     MY_ARRAY;
                          Sum    :    out FLOAT);
   end AdderPkg;
   use CH15_1A.AdderPkg;
   New_Array : MY_ARRAY(FIRST..LAST);


                       -- Implementation of Package1
   package body AdderPkg is separate;

begin
   for Index in New_Array'FIRST..New_Array'LAST loop
      New_Array(Index) := FLOAT(Index);
   end loop;

   Put_Line("Call Add_Em_Up now.");
   Add_Em_Up(New_Array, Sum_Of_Values);
   Put("Back from Add_Em_Up, total is");
   Put(Sum_Of_Values, 5, 2, 0);
   New_Line;

end CH15_1A;




-- Result of execution

-- Call Add_Em_Up now
-- Back from Add_Em_Up, total is   27.00
```

```
                       -- Chapter 15 - Programmming example 1
separate (CH15_1A)
                       -- Implementation of Package1
package body AdderPkg is
   procedure Add_Em_Up(In_Dat : in     MY_ARRAY;
                       Sum    :    out FLOAT) is
   Total : FLOAT;
   begin
      Total := 0.0;
      for Index in In_Dat'FIRST..In_Dat'LAST loop
         Total := Total + In_Dat(Index);
      end loop;
      Sum := Total;
   end Add_Em_Up;
end AdderPkg;
```

2. Remove all **use** clauses from the program named e_c14_p1.ada in the last chapter, and prefix the I/O procedure calls with the proper package names. It should be clear to you that there can be no naming conflicts after this is done.(Solution)

```
                       -- Chapter 15 - Programming example 2
with Ada.Text_IO;

procedure CH15_2 is

   type MY_FIXED is delta 0.01 range 20.0..42.0;
   type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);
   type MY_INTEGER is range -13..323;
   X_Value : FLOAT := 3.14;
   Index   : INTEGER := 27;
   Count   : MY_INTEGER;
   What    : BOOLEAN := TRUE;
   Who     : BOOLEAN := FALSE;
   Size    : MY_FIXED := 24.33;
   Today   : DAY := TUE;

   package Int_IO is new Ada.Text_IO.Integer_IO(INTEGER);
   package Flt_IO is new Ada.Text_IO.Float_IO(FLOAT);
   package Enum_IO is new Ada.Text_IO.Enumeration_IO(BOOLEAN);

   package Fix_IO is new Ada.Text_IO.Fixed_IO(MY_FIXED);
   package Day_IO is new Ada.Text_IO.Enumeration_IO(DAY);
   package New_Int_IO is new Ada.Text_IO.Integer_IO(MY_INTEGER);

begin
                                       -- INTEGER outputs
   Ada.Text_IO.Put("Index is --->"); Int_IO.Put(Index);
              Ada.Text_IO.Put("<---"); Ada.Text_IO.New_Line;
   Ada.Text_IO.Put("Index is --->"); Int_IO.Put(Index,3);
              Ada.Text_IO.Put("<---"); Ada.Text_IO.New_Line;
   Ada.Text_IO.Put("Index is --->"); Int_IO.Put(Index,8);
              Ada.Text_IO.Put("<---"); Ada.Text_IO.New_Line(2);

                                       -- FLOAT outputs
   Ada.Text_IO.Put("Put(X_Value) -------->"); Flt_IO.Put(X_Value);
              Ada.Text_IO.New_Line;
   Ada.Text_IO.Put("Put(X_Value,5) ------>"); Flt_IO.Put(X_Value,5);
              Ada.Text_IO.New_Line;
   Ada.Text_IO.Put("Put(X_Value,5,5) ---->"); Flt_IO.Put(X_Value,5,5);
              Ada.Text_IO.New_Line;
   Ada.Text_IO.Put("Put(X_Value,5,5,0) -->"); Flt_IO.Put(X_Value,5,5,0);
              Ada.Text_IO.New_Line(2);

                                       -- MY_FIXED outputs
   Ada.Text_IO.Put("Put(Size) -------->"); Fix_IO.Put(Size);
              Ada.Text_IO.New_Line;
   Ada.Text_IO.Put("Put(Size,5) ------>"); Fix_IO.Put(Size,5);
              Ada.Text_IO.New_Line;
   Ada.Text_IO.Put("Put(Size,5,5) ---->"); Fix_IO.Put(Size,5,5);
              Ada.Text_IO.New_Line;
   Ada.Text_IO.Put("Put(Size,5,5,0) -->"); Fix_IO.Put(Size,5,5,0);
              Ada.Text_IO.New_Line(2);

                                       -- BOOLEAN outputs
   Ada.Text_IO.Put("What is ---->"); Enum_IO.Put(What);
              Ada.Text_IO.Put("<---"); Ada.Text_IO.New_Line;
   Ada.Text_IO.Put("Who is ----->"); Enum_IO.Put(Who);
              Ada.Text_IO.Put("<---"); Ada.Text_IO.New_Line;
   Ada.Text_IO.Put("What is ---->"); Enum_IO.Put(What,7);
              Ada.Text_IO.Put("<---"); Ada.Text_IO.New_Line;
   Ada.Text_IO.Put("Who is ----->"); Enum_IO.Put(Who,8);
              Ada.Text_IO.Put("<---"); Ada.Text_IO.New_Line;
   Ada.Text_IO.Put("TRUE is ---->"); Enum_IO.Put(TRUE);
              Ada.Text_IO.Put("<---"); Ada.Text_IO.New_Line;
   Ada.Text_IO.Put("FALSE is --->"); Enum_IO.Put(FALSE);
              Ada.Text_IO.Put("<---"); Ada.Text_IO.New_Line(2);

                                       -- Enumeration outputs
   Ada.Text_IO.Put("Today is --->"); Day_IO.Put(Today);
```

```
              Ada.Text_IO.Put("<---"); Ada.Text_IO.New_Line;
   Ada.Text_IO.Put("Today is --->"); Day_IO.Put(Today,6);
             Ada.Text_IO.Put("<---"); Ada.Text_IO.New_Line;
   Ada.Text_IO.Put("Today is --->"); Day_IO.Put(Today,7);
              Ada.Text_IO.Put("<---"); Ada.Text_IO.New_Line;
   Ada.Text_IO.Put("WED is ----->"); Day_IO.Put(WED);
              Ada.Text_IO.Put("<---"); Ada.Text_IO.New_Line;
   Ada.Text_IO.Put("WED is ----->"); Day_IO.Put(WED,5);
              Ada.Text_IO.Put("<---"); Ada.Text_IO.New_Line(2);

end CH15_2;




-- Result of execution

-- Index is --->    27<---
-- Index is ---> 27<---
-- Index is --->      27<---
--
-- Put(X_Value) --------> 3.14000000000000E+00
-- Put(X_Value,5) ------>    3.14000000000000E+00
-- Put(X_Value,5,5) ---->    3.14000E+00
-- Put(X_Value,5,5,0) -->    3.14000
--
-- Put(Size) --------> 24.33
-- Put(Size,5) ------>   24.33
-- Put(Size,5,5) ---->   24.32813
-- Put(Size,5,5,0) -->   24.32813
--
-- What is ---->TRUE<---
-- Who is ----->FALSE<---
-- What is ---->TRUE   <---
-- Who is ----->FALSE   <---
-- TRUE is ---->TRUE<---
-- FALSE is --->FALSE<---
--
-- Today is --->TUE<---
-- Today is --->TUE   <---
-- Today is --->TUE    <---
-- WED is ----->WED<---
-- WED is ----->WED  <---
```

3. Change the name of the two compilation units in e_c15_p1.ada without modifying the filename, and modify e_c15_p2.ada to correspond to the new package name. Use a name that contains more characters than your operating system permits for a filename. Compile each file and link them together to result in an executable program.(Solution 1)(Solution 2)

```
                         -- Chapter 15 - Programming example 3

                 -- Interface of NewAdderPkg
package NewAdderPkg is
   type MY_ARRAY is array(INTEGER range <>) of FLOAT;
   procedure Add_Em_Up(In_Dat : in     MY_ARRAY;
                       Sum    :    out FLOAT);
end NewAdderPkg;
```

```
                 -- Implementation of NewAdderPkg
package body NewAdderPkg is
   procedure Add_Em_Up(In_Dat : in     MY_ARRAY;
                       Sum    :    out FLOAT) is
   Total : FLOAT;
   begin
      Total := 0.0;
      for Index in In_Dat'FIRST..In_Dat'LAST loop
         Total := Total + In_Dat(Index);
      end loop;
      Sum := Total;
   end Add_Em_Up;
end NewAdderPkg;




-- Result of execution

-- (This is not a stand alone package, so it has no output.)



                           -- Chapter 15 - Programming example 3
with Ada.Text_IO, Ada.Float_Text_IO;
use Ada.Text_IO, Ada.Float_Text_IO;

with NewAdderPkg;
use NewAdderPkg;

procedure CH15_3B is

   FIRST : constant := 2;
   LAST  : constant := 7;
   Sum_Of_Values : FLOAT;

   New_Array : MY_ARRAY(FIRST..LAST);

   procedure Summer(In_Dat : MY_ARRAY;
                    Sum    : out FLOAT) renames NewAdderPkg.Add_Em_Up;
begin
   for Index in New_Array'FIRST..New_Array'LAST loop
      New_Array(Index) := FLOAT(Index);
   end loop;

   Put_Line("Call Add_Em_Up now.");
   Add_Em_Up(New_Array, Sum_Of_Values);
   Put("Back from Add_Em_Up, total is");
   Put(Sum_Of_Values, 5, 2, 0);
   New_Line;

          -- The next three statements are identical
   Add_Em_Up(New_Array, Sum_Of_Values);
   NewAdderPkg.Add_Em_Up(New_Array, Sum_Of_Values);
   Summer(New_Array, Sum_Of_Values);

end CH15_3B;




-- Result of execution
-- Call Add_Em_Up now
-- Back from Add_Em_Up, total is   27.00
```

Ada Tutorial - Chapter 16

EXAMPLE PROGRAMS

WHY IS THIS CHAPTER INCLUDED?

It would be a real disservice to you to discuss all of the constructs available with Ada, then simply drop you in the middle of a large program and tell you to put it all together. This chapter is intended to illustrate to you, with simple examples, how to put some of these topics together to build a meaningful program. The examples are larger than any programs we have examined to this point, but they are still very simple, and should be easy for you to follow along and understand what they are doing.

THE CHARACTER STACK

Example program ------> e_c16_p1.ada

```
                                       -- Chapter 16 - Program 1
package CharStak is

procedure Push(In_Char : in CHARACTER);  -- In_Char is added to the
                                         -- stack if there is room.

procedure Pop(Out_Char : out CHARACTER); -- Out_Char is removed from
                                         -- stack and returned if a
                                         -- character is on stack.
                                         -- else a blank is returned

function Is_Empty return BOOLEAN;        -- TRUE if stack is empty

function Is_Full return BOOLEAN;         -- TRUE if stack is full

function Current_Stack_Size return INTEGER;

procedure Clear_Stack;                   -- Reset the stack to empty

end CharStak;





package body CharStak is

Maximum_Size : constant := 25;
Stack_List : STRING(1..Maximum_Size); -- The stack itself, purposely
                                      -- defined very small.
Top_Of_Stack : INTEGER := 0;          -- This will always point to
                                      -- the top entry on the stack.

procedure Push(In_Char : in CHARACTER) is
begin
   if not Is_Full then
      Top_Of_Stack := Top_Of_Stack + 1;
      Stack_List(Top_Of_Stack) := In_Char;
   end if;
end Push;


procedure Pop(Out_Char : out CHARACTER) is
begin
   if Is_Empty then
```

```
      Out_Char := ' ';
   else
      Out_Char := Stack_List(Top_Of_Stack);
      Top_Of_Stack := Top_Of_Stack - 1;
   end if;
end Pop;


function Is_Empty return BOOLEAN is
begin
   return Top_Of_Stack = 0;
end Is_Empty;


function Is_Full return BOOLEAN is
begin
   return Top_Of_Stack = Maximum_Size;
end Is_Full;


function Current_Stack_Size return INTEGER is
begin
   return Top_Of_Stack;
end Current_Stack_Size;


procedure Clear_Stack is
begin
   Top_Of_Stack := 0;
end Clear_Stack;

end CharStak;
```

The program named e_c16_p1.ada illustrates how you can define your own stack for use in your programs. A stack is a list of homogeneous items that grows and shrinks in such a way that the last item entered will always be the first removed, thus it is often referred to as a Last In, First Out (LIFO) list. In order to keep it simple, we will only allow the stack to store **CHARACTER** type variables, although it could be defined to store any type of variables we desired, even arrays or records.

THE SPECIFICATION PACKAGE

The specification package is defined in lines 2 through 20, and gives the user everything he needs to know to use the character stack package. You will see that the user does not know how the stack is implemented, or even how big the stack can grow to unless he looks into the package body, and if we hide the body from him, he has no way of knowing this information. All he can do is use the package the way we tell him it works. In a real system, we would have to tell him how big the stack could grow, because that would probably be valuable information for him when he designed his software, but for our purposes we are not going to let him know the maximum size. Of course, we must then provide a means to be sure that he cannot force the stack to get larger than its maximum size. The procedure **Push** is defined in such a way that if the stack is full and the character will not fit, it is simply not put on the list. The user must check if the stack is full by calling the function **Is_Full** which returns a **BOOLEAN** value of **TRUE** if there is no additional room on the stack.

Careful study of the package specification will reveal that enough functions and procedures are included to allow effective use of the character stack. The package body simply defines the actual implementation required to do the work defined in the specification. Once again, you should be able to understand the package body since there is nothing here which is new or strange to you. It should be pointed out to you that this is actually not a very efficient way to define a stack, because it is limited to a single type, but it serves our purposes well as an illustration. When we study generic packages in part 2 of this tutorial, we will see a much better way to define the stack so that it can be used with additional types. Chapter 33 of part 2 of this tutorial includes this same program rewritten as a generic package which can be used with virtually any data type. You should compile this file in preparation for using it in conjunction with the next example program.

HOW DO WE USE THE CHARACTER STACK?

Example program ------> **e_c16_p2.ada**

```
                                       -- Chapter 16 - Program 2
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;
with CharStak;
use CharStak;

procedure TryStak is

   Example : constant STRING := "This is the first test.";
   Another : constant STRING :=
                  "This is another test and this should not fit.";

   procedure Fill_The_Stack(Input_Line : STRING) is
   begin
      Clear_Stack;
      for Index in 1..Input_Line'LAST loop
         if Is_Full then
            Put_Line("The stack is full, no more added.");
            exit;
         else
            Push(Input_Line(Index));
         end if;
      end loop;
   end Fill_The_Stack;

   procedure Empty_The_Stack is
   Char : CHARACTER;
   begin
      loop
         if Is_Empty then
            New_Line;
            Put_Line("The stack is empty.");
            exit;
         else
            Pop(Char);
            Put(Char);
         end if;
      end loop;
   end Empty_The_Stack;

begin

   Put_Line(Example);
   Fill_The_Stack(Example);
   Empty_The_Stack;

   New_Line;
   Put_Line(Another);
   Fill_The_Stack(Another);
   Empty_The_Stack;
```

```
end TryStak;




-- Result of execution

-- This is the first test.
-- .tset tsrif eht si sihT
-- The stack is empty.
--
-- This is another test and should not fit.
-- The stack is full, no more added.
--  dna tset rehtona si sihT
-- The stack is empty.
```

The example program e_c16_p2.ada uses the character stack included in the last program. Notice the **with** and the **use** clause in lines 4 and 5. These tell the system to get a copy of **CharStak**, and make it available for use in this program. Two string constants are defined for use later, and two procedures are defined to fill the stack, or add a string of characters to the stack one at a time, and to empty it a character at a time. The main program displays each of the string constants on the monitor, then uses the two procedures to reverse the string of characters and output the reversed strings on the monitor. You should study the program until you understand how it works, even though the program itself is not nearly as important to understand as the concept of the separately compiled package. Be sure to compile and run e_c16_p2.ada to see that it really does what you expect it to do.

THE DYNAMIC STRING PACKAGE

Example program ------> **e_c16_p3.ada**

```
                                       -- Chapter 16 - Program 3
-- This is a dynamic string package which can be used as an aid
-- in writing string intensive programs.  Ada only has a static
-- string capability, so this package was written as an example of
-- how the Ada programming language can be expanded.  The basis
-- for this package is the dynamic string available with the
-- Borland International implementation of Pascal, TURBO Pascal.

-- A dynamic string is defined as an array of characters of maximum
-- length of 255.  The dynamic length of the dynamic string is
-- stored in the string location with index = 0, so the dynamic
-- string must be defined with a lower limit of 0 and an upper
-- limit of whatever the desired maximum length of the string is
-- to be.  The subtype STRING_SIZE below limits the string length
-- when it is defined.

-- Put      Outputs a dynamic string to the monitor
-- ConCat   Concatenates two dynamic strings and puts the result
--           into a third dynamic string
-- Copy     Copies a dynamic string to another dynamic string
-- Copy     Copies a static string to a dynamic string
-- Delete   Deletes a group of characters from a dynamic string
-- Insert   Inserts a group of characters into a dynamic string
-- Length   Returns the dynamic length of a dynamic string
-- Size_Of  Returns the static length of a dynamic string
-- Pos      Returns the first location of a dynamic string within
--           another dynamic string

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

```
package DynStrng is

   subtype STRING_SIZE is INTEGER range 0..255;
   type DYNAMIC_STRING is array(STRING_SIZE range <>) of CHARACTER;

-- Put : Display a dynamic string on the monitor.
procedure Put(Input_String : in     DYNAMIC_STRING);


-- ConCat : Concatenation - The First_String is copied into the
--             Result_String, then the Second_String is copied
--             into the Result_String following the First_String.
--             If all characters fit, Result is set to TRUE.
--             If Result_String will not hold all characters,
--             only as many as will fit are copied and Result
--             is set to FALSE.
--             Result = TRUE, complete copy done.
--             Result = FALSE, some or all not copied
procedure ConCat(First_String  : in     DYNAMIC_STRING;
                 Second_String : in     DYNAMIC_STRING;
                 Result_String : in out DYNAMIC_STRING;
                 Result        :    out BOOLEAN);


-- Copy : The String contained in Input_String is copied into
--             the string Output_String.  This procedure is
--             overloaded to include copying from either dynamic
--             strings or static strings.
--             Result = TRUE, complete copy done
--             Result = FALSE, some or all not copied
procedure Copy(Input_String  : in     DYNAMIC_STRING;
               Output_String : in out DYNAMIC_STRING;
               Result        :    out BOOLEAN);
procedure Copy(Input_String  : in     STRING;
               Output_String :    out DYNAMIC_STRING;
               Result        :    out BOOLEAN);


-- Delete : Beginning at First_Position, as many characters as are
--             indicated by Number_Of_Characters are deleted from
--             String_To_Modify.  If there are not that many, only
--             as many as are left are deleted.
--             Result = TRUE, deletion was complete
--             Result = FALSE, only a partial deletion was done
procedure Delete(String_To_Modify     : in out DYNAMIC_STRING;
                 First_Position       : in     STRING_SIZE;
                 Number_Of_Characters : in     STRING_SIZE;
                 Result               :    out BOOLEAN);


-- Insert : The string String_To_Insert is inserted into the string
--             String_To_Modify begining at location Place_To_Insert
--             if there is enough room.
--             Result = TRUE, insert completed in full
--             Result = FALSE, not enough room for full insert
procedure Insert(String_To_Modify : in out DYNAMIC_STRING;
                 String_To_Insert : in     DYNAMIC_STRING;
                 Place_To_Insert  : in     STRING_SIZE;
                 Result           :    out BOOLEAN);


-- Length : Returns the dynamic length of the string defined by
--              String_To_Measure.
function Length(String_To_Measure : in DYNAMIC_STRING)
                  return STRING_SIZE;


-- Size_Of : Returns the static length of the string, the actual
--              upper limit of the string definition.
function Size_Of(String_To_Measure : in DYNAMIC_STRING)
                  return STRING_SIZE;


-- Pos : Position of substring - The string String_To_Search is
--              searched for the string Required_String beginning
--              at Place_To_Start.
--              Result = TRUE, a search was possible
--              Result = FALSE, no search could be made
--              Location_Found = 0, no string found
--              Location_Found = N, start of matching string
procedure Pos(String_To_Search : in     DYNAMIC_STRING;
              Required_String  : in     DYNAMIC_STRING;
              Place_To_Start   : in     STRING_SIZE;
              Location_Found   :    out STRING_SIZE;
              Result           :    out BOOLEAN);

end DynStrng;







package body DynStrng is

               -- The display procedure overloads the existing
               -- Put procedures to output a dynamic string. Note
               -- that the existing Put is used in this new Put.
procedure Put(Input_String : in     DYNAMIC_STRING) is
begin
   for Index in 1..CHARACTER'POS(Input_String(0)) loop
      Put(Input_String(Index));
   end loop;
end Put;





procedure ConCat(First_String  : in     DYNAMIC_STRING;
                 Second_String : in     DYNAMIC_STRING;
                 Result_String : in out DYNAMIC_STRING;
                 Result        :    out BOOLEAN) is
Intermediate_Result : BOOLEAN;
Character_Count     : STRING_SIZE;
Room_Left           : STRING_SIZE;
begin
                       -- Copy the first into the result string
   Copy(First_String,Result_String,Intermediate_Result);
   if Intermediate_Result then
      Character_Count := CHARACTER'POS(Second_String(0));
      Room_Left := Result_String'LAST
                                - CHARACTER'POS(Result_String(0));
      Result := TRUE;
      if Character_Count > Room_Left then
         Character_Count := Room_Left;
         Result := FALSE;
      end if;
      for Index in 1..Character_Count loop
         Result_String(Index + CHARACTER'POS(Result_String(0))) :=
                                         Second_String(Index);
      end loop;
      Result_String(0) :=
             CHARACTER'VAL(CHARACTER'POS(Result_String(0))
                                                + Character_Count);
   else
      Result := FALSE;
   end if;
end ConCat;




               -- This procedure overloads the name Copy to
               -- copy from one dynamic string to another.
procedure Copy(Input_String  : in     DYNAMIC_STRING;
               Output_String : in out DYNAMIC_STRING;
               Result        :    out BOOLEAN) is
Character_Count : STRING_SIZE;
begin
                      -- First pick the smallest string size
   Character_Count := CHARACTER'POS(Input_String(0));
   if Character_Count > Output_String'LAST then
      Character_Count := Output_String'LAST;
      Result := FALSE; -- The entire string didn't fit
   else
      Result := TRUE;  -- The entire string fit
   end if;

   for Index in 0..Character_Count loop
      Output_String(Index) := Input_String(Index);
   end loop;
   Output_String(0) := CHARACTER'VAL(Character_Count);
end Copy;




               -- This routine overloads the copy procedure name
               -- to copy a static string into a dynamic string.
procedure Copy(Input_String  : in     STRING;
               Output_String :    out DYNAMIC_STRING;
               Result        :    out BOOLEAN) is
Character_Count : STRING_SIZE;
begin
                      -- First pick the smallest string size
   Character_Count := Input_String'LAST;
   if Character_Count > Output_String'LAST then
      Character_Count := Output_String'LAST;
      Result := FALSE; -- The entire string didn't fit
   else
      Result := TRUE;  -- The entire string fit
   end if;

   for Index in 1..Character_Count loop
      Output_String(Index) := Input_String(Index);
   end loop;
   Output_String(0) := CHARACTER'VAL(Character_Count);
end Copy;




procedure Delete(String_To_Modify     : in out DYNAMIC_STRING;
                 First_Position       : in     STRING_SIZE;
                 Number_Of_Characters : in     STRING_SIZE;
                 Result               :    out BOOLEAN) is
Number_To_Delete       : STRING_SIZE;
Number_To_Move         : STRING_SIZE;
Last_Dynamic_Character : STRING_SIZE :=
                                CHARACTER'POS(String_To_Modify(0));
begin
                         -- can we delete any at all?
   if First_Position > Last_Dynamic_Character then
      Result := FALSE;
      return;
   end if;
                         -- Decide how many to delete
   if (First_Position + Number_Of_Characters)
                                   > Last_Dynamic_Character then
      Number_To_Delete := Last_Dynamic_Character
                                             - First_Position + 1;
      Result := FALSE;
   else
      Number_To_Delete := Number_Of_Characters;
      Result := TRUE;
   end if;

                  -- find out how many to move back
   if (Last_Dynamic_Character - (First_Position + Number_To_Delete))
                                                             >= 0 then
      Number_To_Move := 1 + Last_Dynamic_Character
                           - (First_Position + Number_To_Delete);
   else
      Number_To_Move := 0;
   end if;

                  -- now delete the characters by moving them back
   for Index in First_Position..
                          (First_Position + Number_To_Move - 1) loop
      String_To_Modify(Index) := String_To_Modify(Index
                                                + Number_To_Delete);
   end loop;
   String_To_Modify(0) :=
            CHARACTER'VAL(CHARACTER'POS(String_To_Modify(0))
                                               - Number_To_Delete);
end Delete;




procedure Insert(String_To_Modify : in out DYNAMIC_STRING;
                 String_To_Insert : in     DYNAMIC_STRING;
                 Place_To_Insert  : in     STRING_SIZE;
                 Result           :    out BOOLEAN) is
Number_To_Add  : STRING_SIZE;
Number_To_Move : STRING_SIZE;
Room_Left      : STRING_SIZE;
```

```
begin
                        -- Can we add any at all?
   if (Place_To_Insert > String_To_Modify'LAST) or
     (Place_To_Insert > CHARACTER'POS(String_To_Modify(0)) + 1) then
      Result := FALSE;
      return;
   end if;
   Result := TRUE;

                        -- How many can we add?
   Number_To_Add := String_To_Modify'LAST - Place_To_Insert + 1;
   if Number_To_Add > CHARACTER'POS(String_To_Insert(0)) then
      Number_To_Add := CHARACTER'POS(String_To_Insert(0));
   end if;

                        -- Find how many to move forward
   Number_To_Move := CHARACTER'POS(String_To_Modify(0))
                                             - Place_To_Insert + 1;
   Room_Left := String_To_Modify'LAST - Place_To_Insert + 1;
   if Room_Left < Number_To_Move then
      Number_To_Move := Room_Left;
   end if;

                        -- Move them forward
   for Index in reverse Place_To_Insert..Place_To_Insert
                                         + Number_To_Move - 1 loop
      String_To_Modify(Index + Number_To_Add) :=
                                           String_To_Modify(Index);
   end loop;
   for Index in 1..Number_To_Add loop
      String_To_Modify(Index + Place_To_Insert - 1) :=
                                           String_To_Insert(Index);
   end loop;

                        -- Increase the length of the string
   String_To_Modify(0) := CHARACTER'VAL(
             CHARACTER'POS(String_To_Modify(0)) + Number_To_Add);
   if CHARACTER'POS(String_To_Modify(0)) > String_To_Modify'LAST
                                                               then
      String_To_Modify(0) := CHARACTER'VAL(String_To_Modify'LAST);
   end if;

end Insert;




               -- This returns the dynamic length of a string
function Length(String_To_Measure : in DYNAMIC_STRING)
                 return STRING_SIZE is
begin
   return CHARACTER'POS(String_To_Measure(0));
end Length;




               -- This returns the static length of a string
function Size_Of(String_To_Measure : in DYNAMIC_STRING)
                 return STRING_SIZE is
begin
   return String_To_Measure'LAST;
```

```
end Size_Of;



procedure Pos(String_To_Search : in     DYNAMIC_STRING;
              Required_String  : in     DYNAMIC_STRING;
              Place_To_Start   : in     STRING_SIZE;
              Location_Found   :    out STRING_SIZE;
              Result           :    out BOOLEAN) is
End_Search            : STRING_SIZE;
Characters_All_Compare : BOOLEAN;
begin
   Location_Found := 0;
                          -- can we search the string at all?
   if (Place_To_Start < CHARACTER'POS(String_To_Search(0))) and
                     (Place_To_Start < String_To_Search'LAST) then
      Result := TRUE;
   else
      Result := FALSE;
      return;
   end if;

                          -- search the loop for the string now
   End_Search := CHARACTER'POS(String_To_Search(0)) -
                         CHARACTER'POS(Required_String(0)) + 1;

   for Index in Place_To_Start..End_Search loop      -- search loop
      Characters_All_Compare := TRUE;
      for Count in 1..CHARACTER'POS(Required_String(0)) loop
         if Required_String(Count) /=
                          String_To_Search(Count + Index - 1) then
            Characters_All_Compare := FALSE;
            exit;         -- exit loop, a character did not match
         end if;
      end loop;
      if Characters_All_Compare then
         Location_Found := Index;
         return;             -- string match found, return location
      end if;
   end loop;  -- end search loop

end Pos;

end DynStrng;
```

The next example program, named e_c16_p3.ada, fulfills a promise made when we were studying strings. At that time, we mentioned that Ada 83 did not have a dynamic string capability but Ada 95 does have several packages that are defined in the ARM as part of the Ada environment. It will be advantageous for you to learn how to use these supplied packages for use in production programs. For the time being, it will be beneficial to spend some time studying the dynamic string package included in e_c16_p3.ada because it illustrates how to use many of the constructs we have studied in this tutorial to this point

The dynamic string package, although implementing a complete version of a dynamic string package, is not submitted as the final word in string packages. In fact, it has a problem which we will describe later in this chapter. Even after we fix the problem, there is a better string package we can use in all of our production programs. It will also be discussed later in this chapter.

One construct is used which we have not yet studied in this tutorial although we did mention it in passing in the last chapter. Line 33 contains an unconstrained array type declaration. Its use is illustrated in lines 9 and 10 of the program named e_c16_p4.ada. The unconstrained array will be studied in part 2 of this tutorial. Other than this one construct, e_c16_p3.ada uses no portion of Ada that we have not yet studied in this tutorial, so it does not take advantage of the advanced constructs of Ada. It is meant to be a teaching aid only, but you could use it in a production program, which you are welcome to do as a user of Coronado Enterprises tutorials. You are granted the right to use any of the included software for whatever purpose you deem necessary, except commercialization of the tutorial itself. You may need to modify a program slightly for your own purposes, and you have permission to do so.

The dynamic string, as defined here, has a maximum length of 255 characters, with the dynamic length stored in the first element of the string which has a subscript of zero. The string must therefore be declared as a **CHARACTER** string with a lower bound of 0, and an upper bound representing the maximum length that the string will ever be asked to store. This definition comes from Borland International's implementation of Pascal, which they market as TURBO Pascal.

THE DYNSTRNG SPECIFICATION

The specification package of **DynStrng** is defined in lines 29 through 116 of the file which is well commented, giving a thorough definition of the package, and describing how to use it. The body is given in lines 124 through 384, and give the actual implementation. Note that some software developers give you only the specification part of the package, which is all you really need to use it, and hide the body from you. If they provide the compiled body, and the specification source, you have all you need to use the package with that specific compiler. Your compiler came with the package **Ada.Text_IO**, a standard package, and your compiler writer almost certainly did not provide you with the source code to the body. The **Ada.Text_IO** package, as defined in the ARM, is only the specification part of the package **Ada.Text_IO.** It can be found in Annex A.10.1 of the ARM.

You should take notice of the fact that we are overloading the name **Put** on line 36 of this package so we can use it to output a dynamic string to the monitor. The system knows which one we wish to use by the type of the actual parameter used. The careful observer will notice that we use the **Ada.Text_IO** version of **Put** in line 132 of the procedure used to overload the name **Put**. Continuing the discussion of overloading, you will see that we define two procedures with the same name in lines 60 and 63, once again depending on the types to select the proper procedure for each call.

Study the **DynStrng** package until you feel you understand it fairly well, then compile the file to prepare for the next program.

USING DYNAMIC STRINGS

Example program ------> e_c16_p4.ada

```
                                        -- Chapter 16 - Program 4
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_text_IO; use Ada.Integer_Text_IO;
with DynStrng; use DynStrng;

procedure TryStrng is

   Try_This : STRING(1..13);
   Name     : DYNAMIC_STRING(0..15);
   Stuff    : DYNAMIC_STRING(0..35);
   Result   : BOOLEAN;
   Neat     : constant STRING := "XYZ";
   Good3    : STRING(1..3);
   Good4    : STRING(1..4);
   Column   : INTEGER;
```

```
begin

   Name(0) := CHARACTER'VAL(3);
   Stuff(0) := CHARACTER'VAL(7);

   Put(Size_Of(Name));
   Put(Size_Of(Stuff));
   Put(Length(Name));
   Put(Length(Stuff));
   New_Line;

   Try_This := "ABCDEFGHIJKL$";
   Copy(Try_This,Stuff,Result);
   Put(Size_Of(Stuff));
   Put(Length(Stuff));
   Put(Stuff); Put(Stuff);
   New_Line(2);

   Copy(Stuff,Name,Result);
   Put(Name); Put(Name); Put(Name); New_Line;

   Concat(Name,Name,Stuff,Result);
   Put(Stuff); New_Line;

   Delete(Stuff,5,3,Result);
   Put(Stuff); New_Line;
   Delete(Stuff,6,3,Result);
   Put(Stuff); New_Line;
   Delete(Stuff,6,3,Result);
   Put(Stuff); New_Line;
   Delete(Stuff,6,3,Result);
   Put(Stuff); New_Line;
   Delete(Stuff,6,3,Result);
   Put(Stuff); New_Line;
   Delete(Stuff,6,3,Result);
   Put(Stuff); New_Line;
   Delete(Stuff,6,3,Result);
   Put(Stuff); New_Line;
   Delete(Stuff,6,3,Result);
   Put(Stuff); New_Line(2);

   Try_This := "1234567890123";
   Copy(Try_This,Stuff,Result);
   Copy(Neat,Name,Result);
   Put(Stuff); Put(Name); New_Line;

   Insert(Stuff,Name,5,Result);
   Put(Stuff); New_Line;
   Insert(Stuff,Name,50,Result);
   Put(Stuff); New_Line;
   Insert(Stuff,Name,2,Result);
   Put(Stuff); New_Line;
   Insert(Stuff,Name,24,Result);
   Put(Stuff); New_Line;
   Insert(Stuff,Name,5,Result);
   Put(Stuff); New_Line;
   Insert(Stuff,Name,5,Result);
   Put(Stuff); New_Line;
   Insert(Stuff,Name,5,Result);
   Put(Stuff); New_Line;
   Insert(Stuff,Name,5,Result);
   Put(Stuff); New_Line(2);

   Good3 := "123";
   Try_This := "1234567890123";
   Copy(Try_This,Stuff,Result);
   Copy(Good3,Name,Result);
   Pos(Stuff,Name,1,Column,Result);
   Ada.Text_IO.Put("Found in column number"); Put(Column); New_Line;
   Pos(Stuff,Name,2,Column,Result);
   Ada.Text_IO.Put("Found in column number"); Put(Column); New_Line;
   Pos(Stuff,Name,7,Column,Result);
   Ada.Text_IO.Put("Found in column number"); Put(Column); New_Line;
   Pos(Stuff,Name,12,Column,Result);
   Ada.Text_IO.Put("Found in column number"); Put(Column); New_Line;
   Pos(Stuff,Name,18,Column,Result);
   Ada.Text_IO.Put("Found in column number"); Put(Column); New_Line;
   Pos(Stuff,Name,50,Column,Result);
   Ada.Text_IO.Put("Found in column number"); Put(Column); New_Line;

end TryStrng;
```

The example program e_c16_p4.ada is designed to use the dynamic string package in various ways by defining strings, inserting characters or strings, deleting portions of strings, and displaying the results. This program was written to test the **DynStrng** package so it does a lot of silly things. There is nothing new or innovative in this utilitarian program, so you will be left on your own to understand, compile, and execute it.

A PROBLEM WITH e_c16_p3.ada

The **DynStrng** package works just fine the way it is used in the **TryStrng** package, but a problem appears when it is used to copy a string constant into a dynamic string. The problem is due to overloading the name **Copy** in lines 60 and 63 of the **DynStrng** specification, and is best illustrated with an example. Consider the following line of Ada code;

```
    Copy("Line of text.", Stuff, Result);
```

In this case, the compiler finds that the string constant could be of type **STRING** or of type **DYNAMIC_STRING**, and does not know which overloading to use, so it gives a compile error saying that it cannot resolve the type. The way to use this package to do this would be to tell the compiler which one to use by qualifying the string constant as shown in this line of code.

```
    Copy(STRING'("Line of text."), Stuff, Result);
```

This will completely resolve the ambiguity and the program will then compile and execute properly. You should include these two lines in a test program to see for yourself that this does resolve the ambiguity.

NOW TO FIX THE PROBLEM

There is an excellent solution to this problem that will render this dynamic string package flexible and useful but it requires the use of a discriminated record which we have not yet studied in this tutorial. In part 2 of this tutorial, we will revisit this dynamic string package and offer a much more flexible package for your information and use.

The **DYNAMIC_STRING** package is a great package for you to study as an illustration of how a package is developed, and how a typical package works. However, it has been superseded by two new packages available for use with any Ada 95 program. The packages named **Ada.Strings.Bounded**, and **Ada.Strings.Unbounded** are parts of the Ada 95 standard library and are available for your use. You should learn their capabilities and limitations thoroughly because they can save you a lot of time.

HOW OLD ARE YOU IN DAYS?

Example program ------> **e_c16_p5.ada**

```
                                       -- Chapter 16 - Program 5
-- This program will calculate the number of days old you are.
-- It is a rather dumb program, but illustrates some interesting
-- programming techniques.  It checks all input to see that they
-- are in the correct range before continuing.  Since the number
-- of days can easily exceed the limits of type INTEGER, and we
-- cannot count on LONG_INTEGER being available, a fixed point
-- variable is used for the total number of days since Jan 1, 1880.
-- This program also passes a record to a procedure, where it is
-- modified and returned.

with Ada.Text_IO, Ada.Integer_Text_IO;
use ADa.Text_IO, Ada.Integer_Text_IO;

procedure Age is

   LOW_YEAR    : constant := 1880;
   MAX         : constant := 365.0 * (2100 - LOW_YEAR);
   type AGES is delta 1.0 range -MAX..MAX;
   Present_Age : AGES;

   package Fix_IO is new Ada.Text_IO.Fixed_IO(AGES);
   use Fix_IO;

   type DATE is record
      Month : INTEGER range 1..12;
      Day   : INTEGER range 1..31;
      Year  : INTEGER range LOW_YEAR..2100;
      Days  : AGES;
   end record;

   Today       : DATE;
   Birth_Day   : DATE;

   procedure Get_Date(Date_To_Get : in out DATE) is
   Temp : INTEGER;
   begin
      Put(" month --> ");
      loop
         Get(Temp);
         if Temp in 1..12 then
            Date_To_Get.Month := Temp;
            exit;                       -- month OK
         else
            Put_Line(" Month must be in the range of 1 to 12");
            Put("                    ");
            Put(" month --> ");
         end if;
      end loop;

      Put("                    ");
      Put(" day ----> ");
      loop
         Get(Temp);
         if Temp in 1..31 then
            Date_To_Get.Day := Temp;
            exit;                       -- day OK
         else
            Put_Line(" Day must be in the range of 1 to 31");
            Put("                    ");
            Put(" day ----> ");
         end if;
      end loop;

      Put("                    ");
      Put(" year ---> ");
      loop
         Get(Temp);
         if Temp in LOW_YEAR..2100 then
            Date_To_Get.Year := Temp;
            exit;                      -- year OK
         else
            Put_Line(" Year must be in the range of 1880 to 2100");
            Put("                    ");
            Put(" year ---> ");
         end if;
      end loop;
      Date_To_Get.Days := 365 * AGES(Date_To_Get.Year - LOW_YEAR)
                  + AGES(31 * Date_To_Get.Month + Date_To_Get.Day);

   end Get_Date;

begin
   Put("Enter Today's date; ");
   Get_Date(Today);
   New_Line;

   Put("Enter your birthday;");
   Get_Date(Birth_Day);
   New_Line(2);

   Present_Age := Today.Days - Birth_Day.Days;
   if Present_Age < 0.0 then
      Put("You will be born in ");
      Present_Age := abs(Present_Age);
      Put(Present_Age, 6, 0, 0);
      Put_Line(" days.");
    elsif Present_Age = 0.0 then
      Put_Line("Happy birthday, you were just born today.");
    else
      Put("You are now ");
      Put(Present_Age, 6, 0, 0);
      Put_Line(" days old.");
   end if;

end Age;
```

The example program named e_c16_p5.ada, is a silly little program, but intended to illustrate how to effectively use the keyboard for input to a program. This program will ask you for today's date, and your birthday, then calculate your age in days. There is no provision for leap year, or even for months with other than 31 days. It is intended to illustrate how to put together an interactive program that could be useful in some way.

Once again, since we have not studied the advanced topics of Ada yet, we have a limited number of constructs to use. The example programs at the end of Part 2 of this tutorial repeats this program, but uses the predefined Ada package **Calendar** to get today's date rather than asking the user to supply it. The advanced topics will add flexibility to your use of Ada. Compile and run this program to get a feel for how to write an interactive program.

PROGRAMMING EXERCISES

1. Add an additional function to e_c16_p1.ada to return a value indicating how many more characters can be pushed onto the stack.(Solution)

```
                              -- Chapter 16 - Programming exercise 1
package CharStak is

function Room_Left return INTEGER;       -- Room left on stack

procedure Push(In_Char : in CHARACTER);  -- In_Char is added to the
                                         -- stack if there is room.

procedure Pop(Out_Char : out CHARACTER); -- Out_Char is removed from
                                         -- stack and returned if a
                                         -- character is on stack.
                                         -- else a blank is returned

function Is_Empty return BOOLEAN;        -- TRUE if stack is empty

function Is_Full return BOOLEAN;         -- TRUE if stack is full

function Current_Stack_Size return INTEGER;

procedure Clear_Stack;                   -- Reset the stack to empty

end CharStak;





package body CharStak is

Maximum_Size : constant := 25;
Stack_List : STRING(1..Maximum_Size); -- The stack itself, purposely
                                      -- defined very small.
Top_Of_Stack : INTEGER := 0;          -- This will always point to
                                      -- the top entry on the stack.

function Room_Left return INTEGER is
begin
   return Maximum_Size - Top_Of_Stack;
end Room_Left;


procedure Push(In_Char : in CHARACTER) is
begin
   if not Is_Full then
      Top_Of_Stack := Top_Of_Stack + 1;
      Stack_List(Top_Of_Stack) := In_Char;
   end if;
end Push;


procedure Pop(Out_Char : out CHARACTER) is
begin
   if Is_Empty then
      Out_Char := ' ';
   else
      Out_Char := Stack_List(Top_Of_Stack);
      Top_Of_Stack := Top_Of_Stack - 1;
   end if;
end Pop;


function Is_Empty return BOOLEAN is
begin
   return Top_Of_Stack = 0;
end Is_Empty;


function Is_Full return BOOLEAN is
begin
   return Top_Of_Stack = Maximum_Size;
end Is_Full;


function Current_Stack_Size return INTEGER is
begin
   return Top_Of_Stack;
end Current_Stack_Size;


procedure Clear_Stack is
begin
   Top_Of_Stack := 0;
end Clear_Stack;

end CharStak;
```

2. Use the new function in e_c16_p2.ada to output a message to the monitor indicating the amount of space remaining on the stack at the end of the **Fill_The_Stack** procedure. (Solution)

```
                     -- Chapter 16 - Programming example 2
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;
with CharStak;
use CharStak;

procedure CH16_2 is

   Example : constant STRING := "This is the first test.";
   Another : constant STRING :=
               "This is another test and this should not fit.";
   Number_Of_Blank_Spaces : INTEGER;

   procedure Fill_The_Stack(Input_Line : STRING) is
   begin
      Clear_Stack;
      for Index in 1..Input_Line'LAST loop
         if Is_Full then
            Put_Line("The stack is full, no more added.");
            exit;
         else
            Push(Input_Line(Index));
         end if;
      end loop;
   end Fill_The_Stack;
   procedure Empty_The_Stack is
   Char : CHARACTER;
   begin
      loop
         if Is_Empty then
            New_Line;
            Put_Line("The stack is empty.");
            exit;
         else
            Pop(Char);
            Put(Char);
         end if;
      end loop;
   end Empty_The_Stack;

begin

   Put_Line(Example);
   Fill_The_Stack(Example);
   Number_Of_Blank_Spaces := Room_Left;
   Put("Room left on stack =");
          Put(Number_Of_Blank_Spaces); New_Line;
   Empty_The_Stack;

   New_Line;
   Put_Line(Another);
   Fill_The_Stack(Another);
   Put("Room left on stack ="); Put(Room_Left); New_Line;
   Empty_The_Stack;

end CH16_2;




-- Result of execution

-- This is the first test.
-- Room left on stack =      2
-- .tset tsrif eht si sihT
-- The stack is empty.
--
-- This is another test and should not fit.
-- The stack is full, no more added.
-- Room left on stack =      0
--  dna tset rehtona si sihT
-- The stack is empty.
```

3. A major programming assignment - The best way to learn Ada is to use it, so the following programming suggestion is given. After studying the package e_c16_p3.ada, put it away and attempt to duplicate it from scratch. You will find that you will use nearly every topic covered in part 1 of this tutorial, and if you get completely stumped, you will have the supplied version of the package to help you over the rough spots. Your goal should be to duplicate the supplied package so closely that the existing program named e_c16_p4.ada can use your new version. If you find e_c16_p3.ada too big for a first step, you may wish to try to duplicate e_c16_p1.ada in the same manner before jumping into the dynamic string effort.