Learning Ada 2012 by writing simple games

Chapter 1

So you want to learn Ada!

This is an informal text that will take you through a series of sample programs for you to learn the rudiments of the language. Ada is used for very serious industrial software requiring high reliability. Learning the basics of Ada programming, however, should be fun and enjoyable, and what better way is there to do this than by creating some amusing games. The text does not cover Ada in full, but gives a taste of the language and by the end of it you should be familiar with the core elements of the language as well as concepts of object oriented programming, concurrency and much more.

There are many books and websites to turn to for more details, and the ultimate authority is the Language Reference Manual (LRM) and the Language Rationale (both available free, online). For more information, please visit www.adaic.com and adacore.com. The Ada LRM is the official language definition. It includes snippets of code that are very useful for learners so if you want to know how a feature works, you should read the LRM, although it is a bit cryptic for beginners.

The approach and the games were inspired by and are based on the book “Beginning C++ Through Game Programming” (Third Edition) by Michael Dawson. Readers are encouraged to buy Michael’s book for his insights into software design and Object Oriented Design and Programming.

This text, much like the language, is a work in progress, so please email comments/suggestions to dclusyd@gmail.com

Ada began in the 80’s, with the first release being Ada 83. This was followed by Ada 95, which added extensive object-oriented capabilities, Ada 2005, which added containers and (currently) Ada 2012, which added tools for security and reliability. We will introduce many of these features with our games. From here on in, Ada will mean Ada 2012.

1.1. Installing Ada and AdaGIDE

To get started, you need a compiler, a program that will take your program and convert it into a form that can run on your computer. There are several free compilers. A list is available at www.adaic.org. For these example programs we will use the free version of GNAT (GNU Ada Translator) for Windows, available from AdaCore, along with the AdaGIDE development environment from Martin Carlise. So, here are some instructions to start you off. These have been tested with all the sample programs under Windows 7, 8, 8.1 and 10. Gnat and the examples also run under linux.

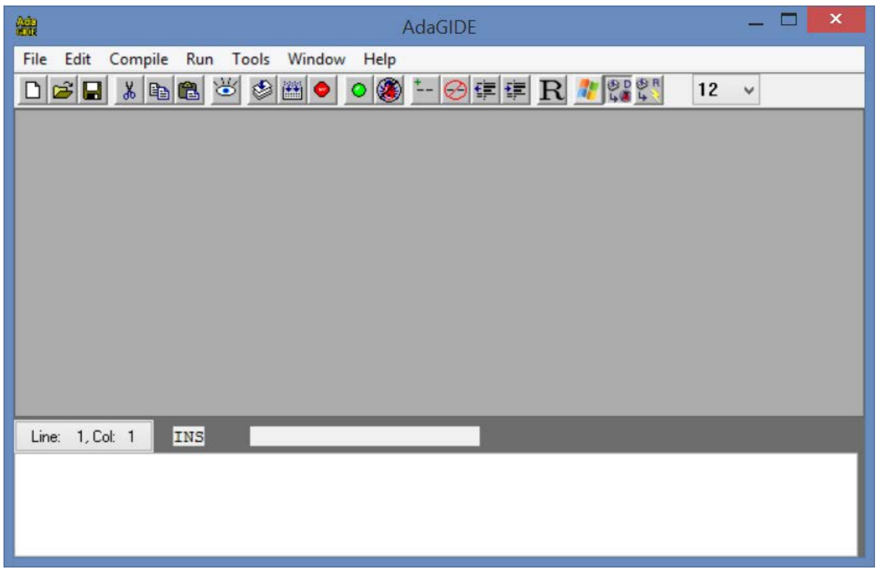
First, get GNAT:

1. Go to libre.adacore.com and click download
2. Select the free version (unless you’d like to buy the pro version)
3. Follow the instructions to choose the current version of GNAT GPL. You don’t need any of the other packages at this point. They are options for later.
4. Click download
5. When it has finished downloading, double click on the downloaded package to install it

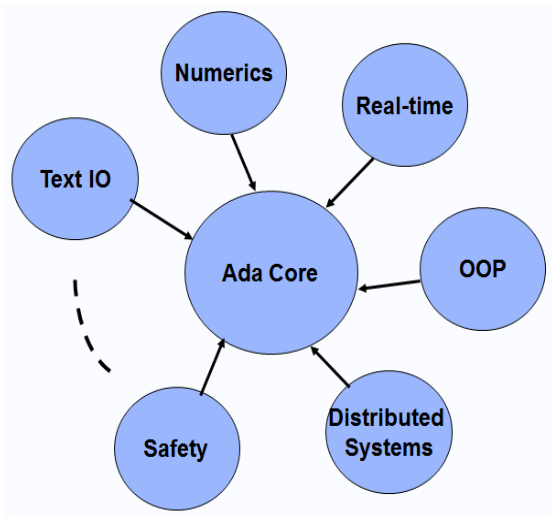
Second, get AdaGIDE for Windows

1. Go to adagide.martincarlisle.com
2. Follow the instructions to download the package
3. Double click on the downloaded package to install it.

Once you have downloaded and installed GNAT and AdaGIDE, double click on AdaGIDE to start it. You should get a window that looks like this:



Some background: Currently, as mentioned above, in 2017 there are four versions of Ada: Ada83, the original design of the language, developed by a French compiler expert, Jean Ichbiah. Ada95 provided a major revision and extension to the language, significantly extending its facilities. Two further revisions followed in 2005 and 2012. In each case, after extensive discussion and consultation, each version was enshrined in a standards document, called the Language Reference Manual (LRM), by the International Standards Organisation (ISO). This makes Ada one of the few computer languages that is thoroughly well defined by an international standard. The version you will have installed is Ada 2012, the latest version, which is upwards compatible with the earlier versions. So, for basic programming, Ada 2012 is essentially the same as Ada 95 and Ada 2005 and all three versions will compile and run most of the programs in this course, except for a few where we will touch on some of the new features of Ada 2012. AdaGIDE includes the LRM for Ada95 in its help menu, so help is just a click away. All the reference manuals are available online at: www.adaic.org/ada-resources/standards/ along with the language rationales, which give the reasoning behind the way in which the language was designed and many guidelines for best use of the language.



Ada is built from a small core, which is suitable for small embedded systems, and is extended by the use of additional **packages** that supplement the core and which aid in the construction of mid-scale to very large systems.

The packages contain software components for all sorts of useful things, such as input-output, numerical calculations, safety and reliability, etc.

Ada programmers often build their own packages, which may contain components that are useful in there are of work, such as games.

Ada2012 Option

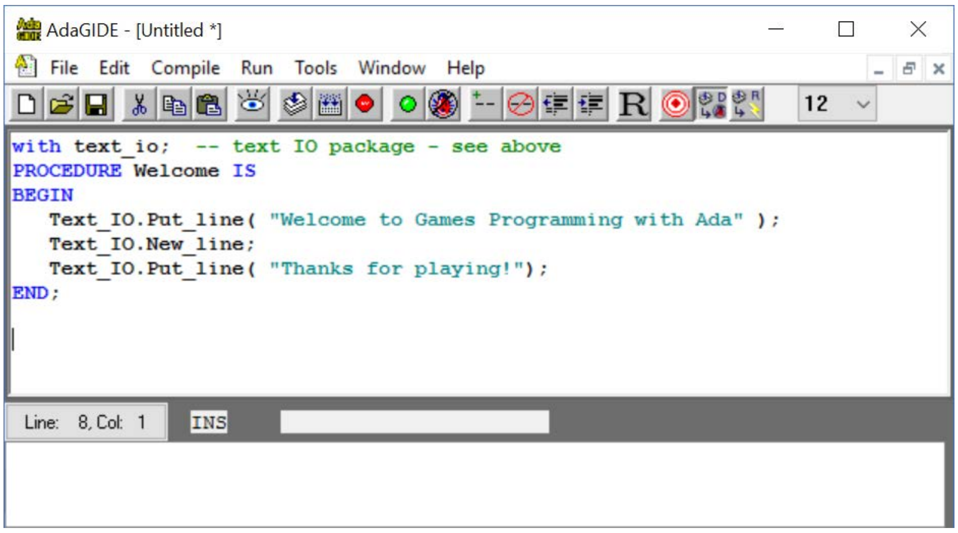
Some of the programs in this tutorial use features only available in Ada2012, so you have to set the Ada2012 option in AdaGIDE in order to compile those programs. If you get an error message “Ada 2012 Feature”, add the compiler option -gnat2012. In the AdaGIDE menu, click Tools, then Project settings, add -gnat2012 in the Compiler Options box and click OK.

1.2 Your first program

Let’s try out a small program. Here is your fist Ada program:

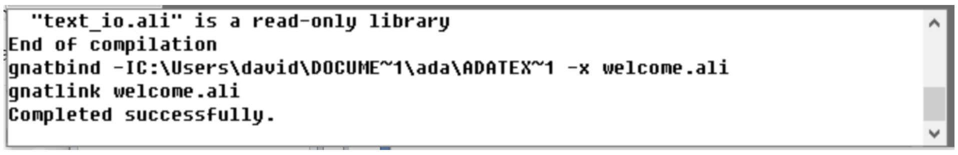
```
with text_io;  -- text IO package – see above
PROCEDURE Welcome IS
BEGIN
   Text_IO.Put_line( "Welcome to Games Programming with Ada" );
   Text_IO.New_line;
   Text_IO.Put_line( "Thanks for playing!");
END;
```

In the AdaGIDE window, click the left-most button, for a new program. The grey screen will turn white, showing that you can enter new code. Then cut and paste the above code into the window, or type it in if you prefer. You should then see:



```
with text_io;  -- text IO package – see above
PROCEDURE Welcome IS
BEGIN
   Text_IO.Put_line( "Welcome to Games Programming with Ada" );
   Text_IO.New_line;
   Text_IO.Put_line( "Thanks for playing!");
END;
```

To prepare the program to run, press function key F3. This is the “Build program” key. This will produce a stream of messages in the lower window, which should end with “Completed successfully.” something like this:

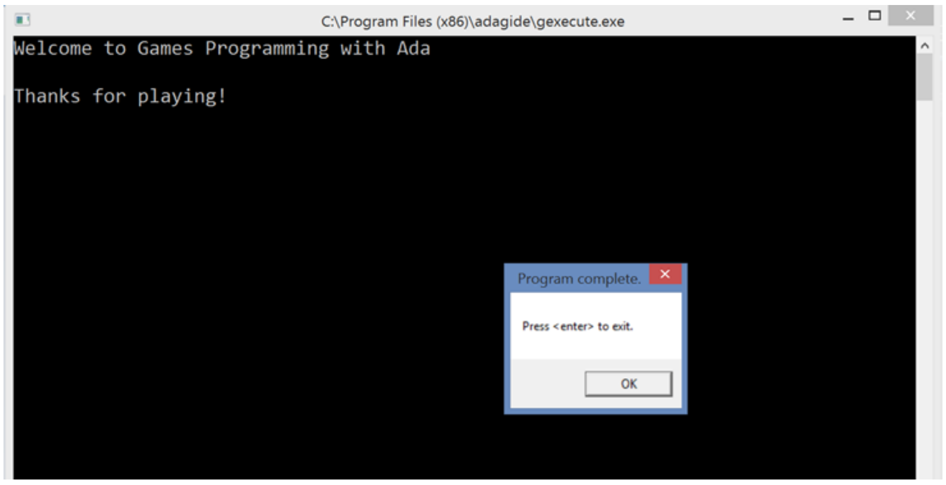


```
  "text_io.ali" is a read-only library
End of compilation
gnatbind -IC:\Users\david\DOCUME~1\ada\ADATEX~1 -x welcome.ali
gnatlink welcome.ali
Completed successfully.
```

You can scroll through the messages with the scroll bar.

If you get the message “Completed successfully.”, your program is ready to run. Press the Function Key F4. This is the Run key. A Command window will open and display the message “Welcome to Games Programming with Ada”, followed by the message “Thanks for playing!”.

Note the small window inside the larger one. Press enter or click the button OK to end the program and return to AdaGIDE.



```
Welcome to Games Programming with Ada

Thanks for playing!
```

As a first step towards your own program, change the message that the program displays to some other message, perhaps something like “Donald says Hi to all brave Ada Games Programmers”. To do this, back in AdaGIDE, just place the cursor on the message in the line:

```
        Text_io.Put( "Welcome to Games Programming with Ada" );
```

delete the existing text and type in your new text, just as you would with any other text editor. Then hit F3, followed by F4, to rebuild the program and run the new version.

Before you do anything else, you should save your program. Click File in the AdaGIDE menu bar:

and then click Save (or Save As). A file folder screen opens. Go to a folder where you want to save your work, and click Save. Note that the program will have the name Welcome of the type Ada File. This is because AdaGIDE requires the file to have the same name as the top procedure of the program, in this case Welcome.

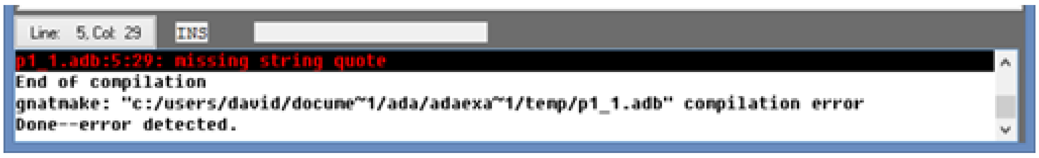
The rest of the buttons in the menu provide facilities to edit, build, run code, switch windows, reformat (tidy up) code, etc. Click Help then Contents to explore the menu options.

1.3 How to deal with errors

Now, let’s suppose that in editing the program, you made a typing mistake or two. What happens when you try to build and run the program? Ada, like all computer systems, is very fussy, even pernickety. It wants everything exactly correct, without a comma or semi-colon out of place. So, let's suppose you have typed in the line

```
            Text_IO.Put( "Angelina say Hi )
```

There are two errors in this line, so AdaGIDE will not build the program. Let's see what happens. Enter the line exactly as above and press F3. The program does not compile and the message window shows the following:



```
Line:  5, Col: 29    INS
p1_1.adb:5:29: missing string quote
End of compilation
gnatmake: "c:/users/david/docume~1/ada/adaexa~1/temp/p1_1.adb" compilation error
Done--error detected.
```

The highlighted line in black with red text in the lower window says "Missing string quote", that is, the messge text needs to end with a quote and doesn't. Note the error message tells you the line and column at which it has decided there is an error. Put your cursor just after Hi and type in the quote, so you have:

```
            Text_IO.Put( "Angelina says Hi” )
```

Now try again (hit F3). We have only fixed one error, so we will get another error message and this time the error message is Missing ";". Put your cursor at the end of the line given in the error message, add the ; and hit F3 and hopefully all will be well!

Sometimes AdaGIDE will detect and report multiple errors. You should then, systematically correct them, starting from the first error, until the build completes successfully.

As you see, the compiler tries to help you by telling what and where it has decided there is an error, but it does not (usually) tell you how to fix it. This you have to figure out for yourself. Often, you will have to refer to the Language Reference Manual (LRM) for details of exactly how a particular feature of the language should be used. For example, you have used text_io, so you might want more details about it. In AdaGIDE, click Help, then Language RM, then put text_io in the search window and you will get many topics relating to text_io. Often the LRM gives little examples of how to write code. For example, in your program, the text "Angelina say Hi" is called a string. Look at the list of topics in section 3 of the LRM, double click on 3.6.3 String Types, scroll down, and you will see some examples of strings.

Now let's look at the program line by line. The first line is

```
      with text_io;
```

This says the program will perform text input or output. Text_IO is a package of code for doing this and offers many functions and tools for reading or displaying text. Put, which you have used above, is one of them. Ada programs use lots of packages and we will study more of them later.

The next line

```
      PROCEDURE Welcome IS
```

names the program as Welcome.

The next line

```
BEGIN
```

marks the start of the program (where it will begin to execute when you press run).

The next line, as you have already found out, sends a message to the display:

```
Text_IO.Put_line( "Welcome to Games Programming with Ada" );
```

The next line skips one line on the display:

```
Text_IO.New_line;
```

You can skip as many lines as you like on the display. E.g. New_line(5) skips 5 lines. New_line(n); skips n lines on the output. New_line on its own, without the number n, skips one line. The n is called a parameter and in this case has a default value (the value it takes if you leave it out) of 1.

The next line displays the farewell message:

```
Text_IO.Put_line( "Thanks for playing!");
```

Finally, the end of the program is marked by

```
END;
```

Games will usually begin with a welcome message and end with a farewell message, as does this example!

1.4 Your second program

Now, as an exercise, write a program that outputs several lines to the screen. You can use as many put and newline statements as you like in a program. Try writing a program to output a simple picture to the display, made up of various characters. Here is an example of a picture of a face drawn with 5 put_line statements:

```
\\\\\\\\\\\
|(*) | (*)|
|    0    |
\_________/
 \xxxxxxx/
```

All the programs in this course are available online in adabookexamples.zip and can be downloaded along with the text. A program to draw the face shown above can be found in the adabookexamples.zip file in DrawFace.ada.

1.5 Numbers

Computers also need to be able to calculate with numbers. In fact this is what they were originally invented for. Our next program will do some calculations with whole numbers, or integers, and display the results. To be able to do this, we need to use the Ada.integer_text_io package in addition to text_IO. I will also add the USE clause, which tells Ada to assume, wherever possible, that I mean text_io or integer_text_io without having to type it in. This can save quite a lot of typing, so I like to do this. Some Ada programmers dislike the use clause and prefer using the full package name. You are free to use the full package name whenever you like if you wish to be sure which one is intended.

With USE clauses

```
with text_io; use text_io;
with ada.integer_text_io; use ada.integer_text_io;

procedure Simple_Calc is
Begin
   Put("5 + 9 - 3 = " ); Put(5 + 9 - 3, 3); new_line;
   Put("7 * 6 / 5 = " ); Put(7 * 6 / 5, 3); new_line;
END Simple_Calc;
```

Without USE clauses

```
with text_io;
with ada.integer_text_io;

procedure Simple_Calc2 is
Begin
   text_IO.Put("5 + 9 - 3 = " );
   ada.integer_text_io.Put(5 + 9 - 3, 3);
   text_IO.new_line;
   text_IO.Put("7 * 6 / 5 = " );
   ada.integer_text_io.Put(7 * 6 / 5, 3);
   text_IO.new_line;
END Simple_Calc2;
```

The main advantage of the USE clauses is cutting down typing.

The main advantage of leaving out the USE clauses is that you have to put the package names into the code wherever they are used, so it is always clear which packages are being used.

The Ada compiler can figure out which packages are intended, so if you build and run either version of the program, its output is:

```
5 + 9 - 3 =  11
7 * 6 / 5 =   8
```

There are several points to note about this program:

1. Ada.integer_text_io is the integer input-output package to display integers. It must be added to our program the same way we did with the text_io package.
2. We can add the USE clauses to reduce the amount of typing. The compiler chooses the correct package depending on whether text or a number is to be displayed.
3. `Put("5 + 9 - 3 = " );` displays a text message, as before. Text messages are indicated by text in quotes, called strings.

4. `Put(5 + 9 - 3, 3);` is different. It computes and then displays a number, in this case the result of the calculation 5 + 9 – 3, so it displays the result of 11. The second number, 3 in this case, tells the integer put procedure to use 3 columns (character spaces) for the output. This can be any number n, and specifies the field width. If left out, its default value is taken as 11 columns.
5. `Put(7 * 6 / 5, 3);` displays a result of 8. The actual calculation is done from left to right and has a mathematical result of 8 remainder 2, but the remainder part is discarded and only the integer part is displayed. Integers are treated by the computer as whole numbers only, no fractional or remainder parts are retained.
6. As in school math, calculations have precedence rules. * and / are done before + and -. Brackets can be used to clarify what is wanted, so 5 + 3 * 8 – 2 is calculated as 5 + (3 * 8) – 2 = 27. If the order were left to right, the result would be (5 + 3) * 8 – 2 = 62, a very different result. Always add brackets if you are not sure.
7. The range of integers, if not specified, is limited on 16 bit computers to -32,768 to + 32,767 and on 32 bit computers to -2,147,483,648 to 2,147,483,647. In Ada, however, you can specify the range of numbers, as we will see later.

1.6 Input

In addition to output from our programs, we need to input data to them. A program with no input is of little use. When data is read in to a program, it must be recorded, or stored, somewhere, so it can be used. The places for storing data are called **variables.**

In Ada, different variables must be used for text (strings) and for numbers like integers. First we will write a program to read a string input.

```
with text_io; use text_io;

procedure String_in is

yourName : string(1..50);   -- declares a string variable
Length : Integer;           -- declares an integer variable

Begin
   Put("Please type in your name: " );
   get_line(yourName, length);
   new_line;
   Put("Welcome to Ada, " ); Put(YourName(1..Length)); New_Line;
   Put("Tell me, "); Put(YourName(1..Length));
   Put(", are you enjoying learning Ada?"); New_Line;
END String_in;
```

The program output is

```
Please type in your name: Dave

Welcome to Ada, Dave
Tell me, Dave, are you enjoying learning Ada?
```

The name Dave above is typed in by the user. This program has two variables.

The first variable is declared as `yourName : string(1..50);` which says it is a string and can hold a string of 50 characters. The second variable is declared as `Length : Integer;` which says it is an integer. It will be used to hold the actual length of the string read in, which can only be up to 50 characters, the size we made the string variable yourName.

From this example, we can see that Ada programs have the general form

```
with <packages>; use <packages>;  -- use is optional
procedure A_Name is         -- name of the program
   <declarations>;          -- declare all variables
Begin                       -- start of executable part
   <executable statements>
END A_Name;                 -- end of program. A_Name is optional
```

Simple strings in Ada are fixed in length. In this example, `yourName : string(1..50)` declares to the program that we want a string that can hold up to 50 characters. The second variable, `length`, is an integer, which is needed by the text input statement, `get_line(yourName, length);` which reads text from the keyboard up to the end of the line, indicated by your hitting enter. It stores the text in the string variable `yourName` and puts into the variable `length` the number of characters you actually typed in, since different names have different lengths.

Having read in the string with the user's name, the next line outputs a messages, made up of two parts, `Welcome to Ada,` followed by the name that the user typed in, in this case, `Dave.` The next two lines output a message made up of three parts, `Tell me, then` the name that the user typed in, followed by `, are you enjoying learning Ada?`

For the name part, only the actual number of characters typed in should be output. The procedure call `Put(YourName(1..Length));` sends only the slice `1..Length` to Put, since that was all that was typed in and so the rest of the declared string does not contain valid characters, just empty space, which may actually contain junk.

Strings in Ada can be manipulated using many functions. Here we have taken a slice of a string. Any slice can be taken of any string, for example, suppose a string "This is a silly joke" were stored in a string variable called jolly, then jolly(2..7) would be the shorter string "his is".

Strings can also be joined together, using the & operator. This is called catenation. So the three part string in the above program could be turned into one long string by joining the 3 parts as follows:

```
   "Tell me, " & YourName(1..Length) & ", are you enjoying learning Ada?"
```

This would allow the above program to be simplified. We could replace the lines

```
   Put("Welcome to Ada, " ); Put(YourName(1..Length)); New_Line;
   Put("Tell me, "); Put(YourName(1..Length));
   Put(", are you enjoying learning Ada?"); New_Line;
```

with

```
   Put("Welcome to Ada, " & YourName(1..Length)); New_Line;
   Put("Tell me, " & YourName(1..Length) &
      ", are you enjoying learning Ada?"); New_Line;
```

By joining the strings we want to output, we can replace 5 put procedure calls with only two, which is obviously more efficient.

Many more things can be done with strings. Some of these will be discussed later.

The above example also included some comments. Comments are very useful in programs to explain to the reader what the program does. Comments always start with a double dash (--), what follows is not compiled or executed, it is explanatory text.

We also see that the program has a neat structure. This helps to make it easy to read. The whole program could be written as:

```
with text_io; use text_io; procedure Welcome is yourName :
string(1..50); Length : Integer; Begin Put("Please type in your
name: " ); get_line(yourName, length); new_line; Put("Welcome to
Ada, " ); Put(YourName(1..Length)); New_Line; Put("Tell me, ");
Put(YourName(1..Length)); Put(", are you enjoying learning Ada?");
New_Line; END Welcome;
```

This makes the program very hard to read and understand.

Good structure and useful comments are very important.

Strings are heavily used in games for the narrative parts of games.

Let’s look at a program that relates a personalised adventure:

```
with text_io; use text_io;

procedure Short_story is

yourName : string(1..50);   -- declares a string variable
Length : integer;

Begin
   Put("Please type in your name: " );
   get_line(yourName, length);
   new_line;
   Put(YourName(1..length));
   Put_Line(" landed on an alien planet and was attacked by 50 aliens.");
   Put(YourName(1..length));
   Put_Line(" fought bravely and killed 12 attackers, after which the");
   Put("remaining aliens fled. ");
   Put(YourName(1..length));
   Put_Line(" was able to crawl back to his ship");
   put_line("where his medical officer successfully treated his wounds.");
END Short_story;
```

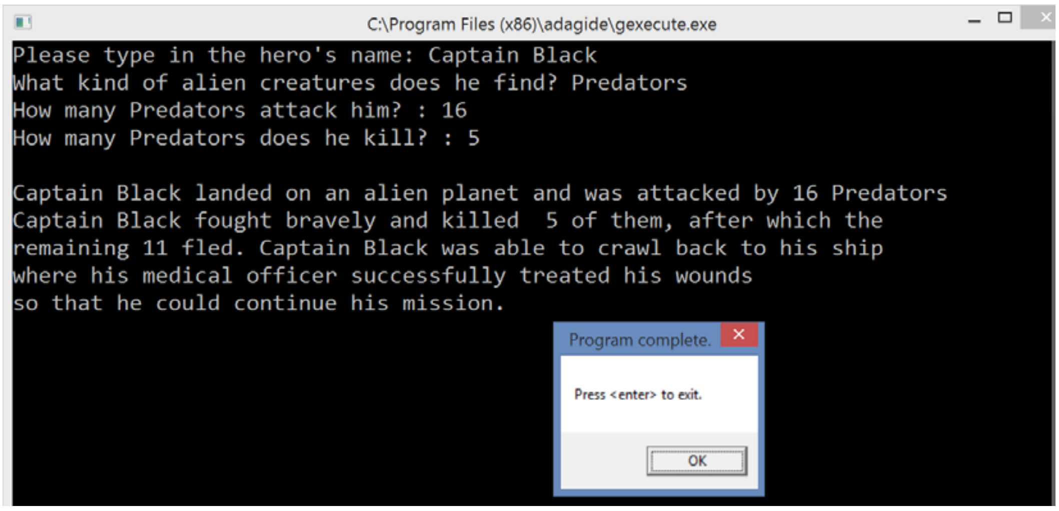
1.6.1 A story with numbers

Now let’s add some numbers and a calculation to our story. We will ask for the type of alien creature, how many of them attack and how many are killed by our hero, to make the story a bit more interesting. We will also simplify the text IO by using Ada’s unbounded strings package. This allows strings to be any length instead of a fixed length.

Here is the code

```
with text_io; use text_io;
with ada.strings.Unbounded;
use ada.strings.Unbounded;
with Ada.Text_IO.Unbounded_IO;
use Ada.Text_IO.Unbounded_IO;
with ada.Integer_Text_IO;
use ada.Integer_Text_IO;

procedure AlienAdventure is

   herosName : Unbounded_String;   -- declares a string variable
   Creatures : Unbounded_String;
   Num_Creatures : Integer;
   Num_killed : integer;

Begin
   Put("Please type in the hero's name: " );
   Get_Line(herosName);
   Put("What kind of alien creatures does he find? ");
   Get_line(Creatures);
   put("How many "); put(Creatures); put(" attack him? : ");
   Get(Num_Creatures);
   Put("How many "); Put(Creatures); Put(" does he kill? : ");
   Get(Num_killed);
   new_line;
   Put(herosName);
   Put(" landed on an alien planet and was attacked by ");
   put(Num_creatures, 2);  -- use 2 spaces for the number (more if needed)
   Put(" "); put(Creatures); new_line;
   Put(herosName);
   Put(" fought bravely and killed "); Put(Num_Killed,2);
   put_line(" of them, after which the");
   Put("remaining "); put(num_creatures - num_killed, 2); put(" fled. ");
   Put(herosName);
   Put_Line(" was able to crawl back to his ship");
   Put_Line("where his medical officer successfully treated his wounds");
   put_line("so that he could continue his mission.");
END AlienAdventure;
```

Compiling and running this program produces the following output:



```
Please type in the hero's name: Captain Black
What kind of alien creatures does he find? Predators
How many Predators attack him? : 16
How many Predators does he kill? : 5

Captain Black landed on an alien planet and was attacked by 16 Predators
Captain Black fought bravely and killed  5 of them, after which the
remaining 11 fled. Captain Black was able to crawl back to his ship
where his medical officer successfully treated his wounds
so that he could continue his mission.
```

Exercises:

a) Add more text to the story
b) Add more variables to the story. E.g. let a team from his crew accompany the hero, specify how many accompany him and how many are killed.

1.7 Real numbers

Integers alone are not enough for serious programs, so Ada provides a number of additional types, including real numbers, decimals and fixed point numbers. For now, we will consider real numbers, i.e. numbers that can be very large, very small, or contain fractional parts. In Ada, these are called floating point numbers, since the decimal point can have varying numbers of digits to the left or right and are declared as **`float.`**

Real numbers are very useful for any kind of number that needs fractions, such as timing fractions of a second, calculations of things like fuel consumption, and so on.

Ada allows the precision and range of floating point numbers to be declared:

```
      Money : float                -- a variable of type float
      Coefficient : float range -1.0..1.0;
```

Floats are represented with two parts, the mantissa (the digits) and the exponent (the power of 10), both of which are signed. Here is an example: 1.601×10^{-19} . Here, the mantissa is 1.601 and is positive, while the exponent is -19 and is negative. This is the electric charge of an electron and is a very small number. It is represented in Ada with the E notation, as 1.601E-19. The E notation avoids writing lots of zeros for very small or very large numbers. Intermediate size numbers can be written with decimal points, as usual, for example temp := 101.3 (degrees Centigrade). Here is a large number: speed_of_light := 2.998E8 (meters per second).

As an example, suppose an intrepid space explorer is travelling to a remote world and need to fuel up the spacecraft, using a formula based on the weight of spacecraft and the distance in lightyears:

```
with Text_Io; use Text_Io;    -- text IO library package
with Ada.Float_Text_IO;
use Ada.Float_Text_IO;              -- Real IO library package

PROCEDURE FuelCalc IS
   Weight, Distance, Fuel : float;  -- declare three floats
begin
   New_Line;
   Put_Line( "Calculate fuel load for spacecraft." );
   New_line;
   Put( "What is the craft loaded weight in tonnes? " );
   Get( Weight );             -- input a real
   New_Line;
   Put( "What is the length of this trip in light years? " );
   Get( Distance );           -- input a real
   Fuel := 100.0 + Weight * 2.0 * (Distance – 1.0) * 0.1;
   New_Line;
   Put( "Fuel load in tonnes = " );
   Put( Fuel, Aft => 1, Exp => 0 ); -- output a real
   New_Line(2);
   Put_Line( "Done. Thanks for visiting." );
End FuelCalc;
```

When we run this program, we get:

```
Calculate fuel load for spacecraft.

What is the craft loaded weight in tonnes? 247

What is the length of this trip in light years? 14.2

Fuel load in tonnes = 752.1

Done. Thanks for visiting.
```

In this program, we 'with' and 'use' the real IO library package, then we declare three float variables, Weight, Distance and Fuel. Next we display a message to tell the user what the program is doing, ask for the weight, then read a float, which gets stored in the float variable Weight. Next we display a message to ask for the distance, then read a float, which gets stored in the float variable Distance. Next we convert the temperature to Celsius, using the formula **100.0 + Weight** * 2.0 ***** (**Distance** – 1.0) *** 0.1;** and save the result in the float variable Fuel. We put decimal points in the constants in the formula because this is a floating point calculation, not an integer calculation and Ada does not allow integers and floats to be mixed as they are different types of variables. Finally we display the result. Floating point output is usually displayed with the E notation, so here we say Exp => 0 to tell the output procedure not to use the E notation. We can also specify how many columns we want before and after the decimal point, with parameters Fore and Aft. In this example, we only use Aft=>1 to use a single decimal point.

Floats normally have about 6 digits of precision, and this may not be enough for some high precision applications, such as global positioning, or inter-bank transactions of large sums of money. For this reason, Ada defines a long_float that has a precision of at least 11 digits, if the hardware supports it.

1.8 Simulating a Javelin Throw

Let's now design a game that uses floats. We will simulate throwing a javelin. Many interesting games involving throwing, shooting, flying, etc. and these require some knowledge of the physics of flights and trajectories. In this case, it is easy to look up the formula for the flight of a javelin, e.g. on Wikipedia. To simplify matters, we assume there is no wind and the effect of air friction can be ignored. We will also assume that the throw is at the optimal angle of 45 degrees, which gets the greatest distance. The formulae are then, with v being the velocity of the throw:

    Distance thrown = v * v / G   FlightTime := 2.0 * v * sin(45) / G;

where G is the acceleration due to gravity, = 9.81 m/s and sin(45) = 0.7071
Advanced programmers will want to read one of the several books available on the maths and physics of games programming. We will also do some timing, as many interesting games involve timing, where a player must make a move within some time, or the timing of a play affects the outcome, or a game is played faster at higher levels.

We will time the player's run-up to calculate a velocity (v) for the throw. We will simulate a "run-up" by timing the entry of 20 characters on the keyboard, the faster the character entry, the faster the run-up and the higher the velocity of the throw. The design of the program is

```
Time 20 keystrokes
If the run-up is too slow, cancel the throw
otherwise calculate the velocity from the run-up time, then
calculate and display the distance thrown and the flight time.
```

This kind of program design is call pseudo-code and sets out what the program will do without all the details needed by the programming language.

To calculate the run-up time, we will use the clock function provided as part of the Ada calendar package. Ada also has a special type, Duration, for a time period. We can calculate the duration of the run-up as the difference between the start and stop times of the input. Then we will calculate the velocity from the run-up time as (10 – run-up time) * 2. We could do this in many ways, but this is fine for this example.

Here is the example:

```
-- ADA floats example - throwing a javelin

WITH Text_Io; USE Text_Io;
WITH Calendar; USE Calendar;
WITH Ada.Float_Text_IO; USE Ada.Float_Text_IO;

PROCEDURE Javelin1 IS
   G      : CONSTANT := 9.81;  -- Acceleration due to gravity
   Stop,
   Start  : Time;  -- Time is the time of day, defined in Calendar
   Taken  : Duration;
   Char   : Character;
   Velocity, Distance, FlightTime : Float;
   Runup : String(1..20);
   length : integer;
BEGIN
   Put("Hit any key to start, then any key 20 times, then Enter,” );
   Put_Line(" to time your runup");
   Get_Immediate(Char); -- gets one character immediately it is struck
   Start := Clock;
   get_line(runup, length);   -- now get the runup characters
   Stop := Clock;
   Taken := Stop - Start;     -- time taken for runup
   New_Line;
   Put("You took ");
   Put(Float(Taken), Fore => 1, Aft => 2, Exp => 0);
   Put_Line(" seconds for your runup");
   Put("You threw the javelin at ");
   Velocity := (10.0 - Float(Taken)) * 2.0;
   Put(Velocity, Fore => 1, Aft => 2, Exp => 0);
   Put_Line(" meters per second");
   Distance := (Velocity ** 2) / G;
   FlightTime := 2.0*Velocity*0.7071/G;
   Put("You threw it ");
   Put(Distance, Fore => 1, Aft => 2, Exp => 0);
   Put_Line(" meters.");
   Put("It flew for ");
   Put(FlightTime, Fore => 1, Aft => 2, Exp => 0);
   Put_Line(" seconds.");
END;
```

The program starts in the usual way, by ‘with’ing the packages we want to use. Since we are going to use a timer, we also ‘with’ the calendar package, which includes a time of day clock, along with functions to handle time and dates, etc. along with the system clock.

Next we define some data. G is a constant, the acceleration due to gravity, which we need in our formulae to calculate the velocity and distance of the throw. Next we define two variables, start and stop, for the times the player's run-up starts and stops. These are of type time, which are part of the calendar package and can store a date and time of day, such as 4 Apr 2015 8:21:43.576. The time resolution depends on the hardware but is accurate to milliseconds on most systems. We will read the time at the start and stop of the run-up and calculate the time taken for the run-up as stop – start. This will be stored in variable taken, of type duration, which can hold a time interval, which Ada calls a duration. The calendar package defines the difference between two times as a duration, so this makes sense. Note that we cannot add two times, eg 7am + 2pm – the calendar package defines this as an illegal calculation, since it does not make sense physically. We can add a time and a duration, which gives a result of type time eg 5 am + 2 hrs (= 7 am). We then have some variables for the velocity, distance, flightTime and the player's inputs for one character to start the run-up, a line of characters to represent the actual run-up and the length of the run-up (in characters.

The program then gives the player some instructions, then waits for a single character to start the run-up. Next start timing the run-up by reading the current time. We read a line of characters, ending with Enter. We then read the time again, after the run-up, and calculate the time taken as taken = stop – start and tell the player how long the run-up took. We then use our formulae to calculate the velocity, distance and flight time and report these results to the player.

This little example shows some calculations with floats and with time and duration, but it has two major problems – the user can cheat by entering a short line of less than 20 characters, or could enter the line too slowly, so that the run-up time taken is more than 10 seconds, which results in a negative velocity for the throw.

1.9 Ada types and subtypes

Ada likes users to define specific types for new purposes. This can help a lot to make the purpose of a program clear and prevent errors. For example, one can make a type money, a type distance, and type area, etc. Ada even has special types of time and duration, as we saw above. Here are some examples:

```
type fuelT is range 0.0 .. 1.0e6;  -- define two types. These are subtypes
type photonEnergyT is digits 6 range 0.0 .. 1.0e10; -- of float
type basketBallScoreT is range 0 .. 200;  -- subtype of integer
type passengersT is range 0 .. 500;

cruiserFuel, fighterFuel : fuelT; -- now make some variables
fighterEnergy :  photonEnergyT;   -- of those types
gameScore : basketBallScoreT
cruiserCapacity : passengersT := 176;  -- make and initialize variable

fighterFuel := 1.0e4;               -- here are some operations with them
cruiserFuel := FighterFuel * 10.0;

cruiserCapacity := 600;                  -- No, out of range

fighterFuel := fighterEnergy;            -- No, can't mix types,

fighterFuel := fighterFuelT(fighterEnergy;); -- but you can explicity convert

gameScore := gameScore - 1.5;  -- Ooops, no, can't mix integers and reals
```

```
TYPE Money IS DELTA 0.01 digits 9 RANGE 0.0 .. 1000.0; ]
PACKAGE Money_IO IS NEW Text_IO.decimal_IO(Money);  -- package for money IO
USE Money_IO;

Fare, Penalty : money;        -- Make some variables of type money
```

Enumeration types are lists of items with specific names or values listed. For example

```
      type Primary_Colours is (red, blue, yellow);
      type Traffic_Light is (green, amber, red);
```

We can then declare variables of these types, and optionally initialise them:

```
      myColours : Primary_colours := yellow;
      NS, EW : Traffic_Light := red;
```

Games programmers should always use types specific to the current action, e.g. a scores type for scores, money type for money. This is good practice and avoids lots of common programming errors where unlike items are accidentally mixed.

Programs often need constants. An example above is the declaration of G, the acceleration due to gravity on earth. Another example might be to declare the maximum number of players in a game, the maximum number of enemies a player can face at one time or the maximum number weapons a player can carry.

Constants are declared with the word CONSTANT and the type of the constant is derived from the object assigned to it. Examples are

```
  DOZEN : CONSTANT := 12  -- integer constant (since 12 is an integer)
  G     : CONSTANT := 9.81;  -- real constant (since 9.81 is a real)
  GROSS : CONSTANT := 12 * DOZEN;  -- assigned value may be calculated
```

String constants, however, must be declared as strings:

```
  REPLY    : CONSTANT string := “Yes or No”  -- string consstant
```

Constants cannot be changed by the program. Any attempt to do so will result in an error message.

2. Tests, conditions and loops

A key ability from which computers derive their power is their ability to make decisions based on testing of conditions. Conditions can be combined with choices in order to make decisions. The basic form for this is the IF statement

2.1 IF statements

The basic form of the IF statement is

```
If <condition> Then statement;
End If;
```

In this form, the statement is executed only if the condition is true, and not executed otherwise. The statement can also be a block of statements, starting with Begin and ending with End.

```
WITH Text_IO; USE Text_IO;
WITH Ada.Integer_Text_IO; USE Ada.Integer_Text_IO;

PROCEDURE Divisible IS
   Number, divisor : integer;
BEGIN
   Put_line( "Check if number is divisible by divisor" );
   Put(“Enter number: ”); get(number);
   Put(“Enter divisor: “); get(divisor);
   New_line;
   If (number rem divisor) = 0 then  -- remainder = 0?
      begin
         put(“Yes,”); put(number); put(“ is divisible by”);
         put(divisor);
         new_line;
      end;
   end if;
 END;
```

Exercise: This program does not have a proper end message. Add one.

As another example of an IF statement, let’s check if a URL (i.e. a web address) contains a particular component, e.g. does http://www.adacore.com contain the substring core. To do this we can use the string function col := index(myString, Pattern); which searches myString to see if it contains Pattern. If it does, col is set to the character number where Pattern starts, otherwise it is set to zero.

Suppose we have initialised the variable URL as follows:

```
URL : string := “http://www.adacore.com”;
```

We can write

```
IF index(URL, “core”) /= 0 then <statement>
```

In the above, the Then part of the If will be executed, because the URL does contain “core” so index returns a non-zero value of 15, which is where the substring “core” starts in this URL.

The complete set of relational tests in Ada is

=	Equal
/=	Not Equal
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal to

Here are some examples. Given X = 3 and Y = 5

X = Y	(False)
X /= Y	(True)
X < Y	(True)
X <= Y	(True)
X > Y	(False)
X >= Y	(False)

The result of a relational test is True or False, and this is called a Boolean (or logical) result, named after George Boole, a mathematician who developed Boolean algebra, a form of mathematical logic.

Boole defined three logical operators, AND, OR and NOT:

A AND B is True if both A and B are True, otherwise it is False
A OR B is True if either A or B or both are True, otherwise it is False
NOT A is True if A is False and False if A is True

In Ada, you can use the logical operators in relational expressions. Here are some examples. Given X=3 and Y=5:

```
X < 10 AND Y > 1   (True)                X < Y OR Y > 10  (False)
NOT (X < 10)       (False)
```

In Ada, you can also use a membership test, IN, using a Range or a List. For example:

```
X IN 1..10      (True)   (Range)         X IN (1 | 3 | 5 | 7)    (True)  (List)
Weapon IN (Sword | Bow | Pike)    (List)
```

Lists are constructed by listing the items, separated b '|'. The items must all be the same type.

Membership tests, relational and logical operators can all be combined:

```
X   NOT IN 1..10  (False)       X = 3 AND Y = 5    (True)       X IN 1..10 AND Y /= 5  (True)
```

IF statements can contain many alternatives, using ELSIF and ELSE. Here is an example of a piece of code, showing a more complex IF statement, part of a program to simulate driving a car:

```
if  Stop_Light = RED then
   Stop;
elsif Stop_Light = GREEN then
   Look_Both_Ways; Go;
elsif Stop_Light = YELLOW then
   Close_Eyes; Go_Fast;
else
   Stop; Look_Both_Ways; Go;
end if;
```

Ada 2012 allows conditional expressions, a kind of shorthand for IF statements, which take the form as shown by the following examples:

```
S := (if N > 0 then +1 else 0);

Put(if N = 0 then "none" elsif N = 1 then "one" else "lots");

Stop_Light := (if Stop_Light = GREEN then YELLOW else RED);
```

Now let's extend the javelin example to handle the input errors of too short or too slow a run-up. We will add the conditions that if the length of the input is less than 20, reject it, and if the run-up takes longer than 10 seconds, reject it. In both cases, we will output an appropriate message. Here is the new version of the program:

```
-- ADA floats example - throwing a javelin

WITH Text_Io; USE Text_Io;
WITH Calendar; USE Calendar;
WITH Ada.Float_Text_IO; USE Ada.Float_Text_IO;

PROCEDURE Javelin2 IS
   G       : CONSTANT := 9.81;  -- Acceleration due to gravity
   Stop,
   Start   : Time;
   Taken   : Duration;
   Char    : Character;
   Velocity, Distance, FlightTime : Float;
   Runup : String(1..20);
   length : integer;
BEGIN
   Put_Line("hit any key to start, then any key 20 times, then Enter, to
time your runup");
   Get_Immediate(Char); -- gets one character immediately it is struck
   Start := Clock;
   get_line(runup, length);
   Stop := Clock;
   Taken := Stop - Start;
   New_Line;
   IF (Taken > 10.0) THEN
      Put_Line("This is not a valid throw - too slow");
   ELSIF (length < 20) THEN
      Put_Line("This is not a valid throw - short run-up");
   ELSE
      BEGIN
         Put("You took ");
         Put(Float(Taken), Fore => 1, Aft => 2, Exp => 0);
         Put_Line(" seconds for your runup");
         Put("You threw the javelin at ");
         Velocity := (10.0 - Float(Taken)) * 2.0;
         Put(Velocity, Fore => 1, Aft => 2, Exp => 0);
         Put_Line(" meters per second");
         Distance := (Velocity ** 2) / G;
         FlightTime := 2.0*Velocity*0.7071/G;
         Put("You threw it ");
         Put(Distance, Fore => 1, Aft => 2, Exp => 0);
         Put_Line(" meters.");
         Put("It flew for ");
         Put(FlightTime, Fore => 1, Aft => 2, Exp => 0);
         Put_Line(" seconds.");
      END;
   END IF;
END;
```

Note the use of the ELSIF which allow us to output a different message for the two conditions.

As an exercise, add another ELSIF to disallow a run-up pf more than 20 characters.

2.2 More choices: the Case statement

The case statement is useful when there are lots of choices. It is makes for a clearer program than a long if statement. For example, the traffic light example can be rewritten with a case statement:

```
     if  Stop_Light = RED then
          Stop;
     elsif Stop_Light = GREEN
     then
          Look_Both_Ways; Go;
     elsif Stop_Light = YELLOW
     then
          Close_Eyes; Go_Fast;
     else
          Stop; Look_Both_Ways;
     Go;
end if;
```

```
case  Stop_Light  is
  when RED    => Stop;
  when GREEN  => Look_Both_Ways;
     Go;
  when YELLOW => Close_Eyes;
     Go_Fast;
  when OTHERS => Stop;
     Look_Both_Ways; Go;
end case;
```

Here is a case statement that might be part of a game of 21 (or blackjack):

```
case Card_total is
    when 1..16  => Draw;
    when 17..20 => Stick;
    when 21     => Twenty_one;
    when 22..31 => Bust;
    when others => bad_total;
end case;
```

Notice that we can use ranges when specifying the cases. The keyword “others” is very useful for handling unspecified cases tidily. Now let’s look at an example program with a case statement:

```
with text_io; use text_io;
PROCEDURE Colours IS
   Choice : Character;
BEGIN
   Put_Line( "Displays Complementary Colors " );
   Put_line( "Enter the first letter of the primary colour." );
   Put( "B.lue, G.reen, O.range, P.urple, R.ed, Y.ellow, Q.uit? " );
   Get_Immediate( Choice );  -- returns the pressed key immediately
   CASE Choice IS
        when 'B' | 'b' => Put_line( "Orange" );
        when 'G' | 'g' => Put_line( "Red"    );
        when 'O' | 'o' => Put_line( "Blue"   );
        when 'P' | 'p' => Put_line( "Yellow" );
        when 'R' | 'r' => Put_line( "Green"  );
        when 'Y' | 'y' => Put_line( "Purple" );
        when 'Q' | 'q' => NULL;
        when others => put_line( "I don't have that colour. Try again" );
   END CASE;
   put_line( "Bye." );
END;
```

As you can see, the case statements show very clearly what we want the program to do. The sort of structure is commonly used in games and other programs to display menus of choices to the user. The NULL says do nothing, so the case statement ends with no message and the program then ends.

Here is another example of the use of a Case statement. We will calculate the day of the week for any given date using a very clever formula called Zeller's Congruence. In historical games, this is a very useful function.

Our program will read in the date as three integers, for the day, month and year, evaluate Zeller's Congruence, and display the result. Here's the basic design of the program:

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use  Ada.Text_IO, Ada.Integer_Text_IO;
procedure Weekday is
    Day    : Integer;
    Month  : Integer;
    Year   : Integer;
    DayOfWeek : Integer;
begin
    Put ("Enter a date as day month year: ");
    Get (Day);
    Get (Month);
    Get (Year);
    -- Apply Zeller's Congruence
    Put (DayOfWeek);
end Weekday;
```

Here is Zeller's congruence:

Day = ((26M-2)/10 + D + Y + Y/4 + C/4 - 2C) mod 7

In the formula, M is the number of the month, D is the day, Y is the last two digits of the year number and C is the century (the first two digits of the year number). Integer division is used, so that 19/4 = 4 rather than 4.75. The mod operation gives the remainder of an integer division, in this case the remainder from dividing by 7, i.e. a value between 0 and 6.

The formula, however, requires that the month number starts with March as month 1, with January and February treated as months 11 and 12 of the previous year. We therefore need to adjust the month and year like this:

```
if Month < 3 then
   Year  := Year - 1;       -- subtract 1 from year number
   Month := Month + 10;     -- convert 1 and 2 to 11 and 12
else
   Month := Month - 2;      -- subtract 2 from month number
end if;
```

The formula gives a result between 0 and 6, where 0 means Sunday and 6 means Saturday. Let's see how this works using February 14th, 1847 as an example. The value of D is 14 and C is 47. February 1847 has to be converted to month 12 of 1846, so M is 12 and Y is 46. This gives us:

Day = ((26M-2)/10 + D + Y + Y/4 + C/4 - 2C) mod 7
= ((26*12-2)/10 + 14 + 46 + 46/4 + 18/4 - 2*18) mod 7
= (310/10 + 14 + 46 + 11 + 4 - 36) mod 7
= (31 + 14 + 46 + 11 + 4 - 36) mod 7
= 70 mod 7
= 0 (Sunday).

It will help to introduce an extra variable for the value C since it's used twice in the formula above.

Here's a complete version of the program:

```
WITH Ada.Text_IO, Ada.Integer_Text_IO;
USE  Ada.Text_IO, Ada.Integer_Text_IO;

PROCEDURE Weekday IS
   Day       : Integer;
   Month     : Integer;
   Year      : Integer;
   Century   : Integer;
   DayOfWeek : Integer range 0..6;
BEGIN
   Put ("Enter a date as day month year: ");
   Get (Day);
   Get (Month);
   Get (Year);
   IF Month < 3 THEN
      Year  := Year - 1;
      Month := Month + 10;
   ELSE
      Month := Month - 2;
   END IF;
   Century := Year / 100;         -- first two digits of Year
   Year    := Year mod 100;       -- last two digits of Year
   DayOfWeek := (((26*Month - 2)/10 + Day + Year + Year/4
         + Century/4 - 2*Century) mod 7);
   Put("That Date is a ");
   CASE DayOfWeek IS
      WHEN 0 => Put_Line("Sunday");
      WHEN 1 => Put_Line("Monday");
      WHEN 2 => Put_Line("Tuesday");
      WHEN 3 => Put_Line("Wednesday");
      WHEN 4 => Put_Line("Thursday");
      WHEN 5 => Put_Line("Friday");
      WHEN 6 => Put_Line("Saturday");
   END CASE;
END Weekday;
```

Notice that the value of Century must be calculated after the if statement. This is because the if statement might change the value of Year; January 2000 is treated as being month 11 of 1999, so the value for Century will be 19 until March 2000. Likewise Year only gets trimmed to its last two digits after the first two digits have been extracted into Century. The order of these calculations is critical to the correct operation of the program.

Also note that we have assumed the user will enter a valid date. An entry like 57 83 -12 will produce a result that will be nonsense. As programmers sometimes say, garbage in, garbage out! Later we will see how to check if the inputs are valid.

We can also modify the program to use an enumeration statement instead of a case statement. We define: type dayOfWeekT is (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday); and display this type directly. The Ada system automatically numbers enumeration types from zero, and an integer is then converted to an enumeration type using the attribute ‘val, as in DayOfWeek := DayOfWeekT’val(num); The textual representation of the enumeration type may then be displayed using the attribute ‘image, as in DayOfWeekT'image(dayofweek). All types in Ada have attributes, such as min, max, val, image, etc. The attributes for every type are listed in the LRM. The ‘image attribute is often used for numerical outputs. Here is the modified program:

```
WITH Ada.Text_IO, Ada.Integer_Text_IO;
USE  Ada.Text_IO, Ada.Integer_Text_IO;
```

```
PROCEDURE Zeller IS
   TYPE DayOfWeekT IS (Sunday, Monday, Tuesday, Wednesday, Thursday,
                       Friday, Saturday);
   Day       : Integer;
   Month     : Integer;
   Year      : Integer;
   Century   : Integer;
   DayOfWeek : DayOfWeekT;
BEGIN
   Put ("Enter a date as day month year: ");
   Get (Day);
   Get (Month);
   Get (Year);
   IF Month < 3 THEN
      Year  := Year - 1;
      Month := Month + 10;
   ELSE
      Month := Month - 2;
   END IF;
   Century := Year / 100;        -- first two digits of Year
   Year    := Year mod 100;      -- last two digits of Year
   DayOfWeek := DayofWeekT'Val((((26*Month - 2)/10 + Day + Year + Year/4
         + Century/4 - 2*Century) mod 7));
   Put("That date is a ");
   put_line(DayOfWeekT'image(dayofweek));
END Zeller;
```

2.3 Loops

So far, we have written a few simple programs that do one thing, then stop. One of the key things that give computers great power is their ability to do the same thing many times, such as calculate wages for all the employees of a large company.

To do this, requires the means to program and manage repetition. Repeated segments of code are called loops.

2.3.1 The basic loop

The form of the basic loop is

```
   loop                                     -- start of loop
      <some statements>
     exit when <some condition is true>     -- loop exit
      <some more statements>
   end loop;
```

The exit statement may be placed anywhere in the loop.

This is a program for converting temperature from Fahrenheit to Celsius that loops, repeatedly asking for more input, until the user enters a value of zero, whereupon it exits. The program uses a simple formula for calculating the conversion. Here is the code:

```
with Text_Io; use Text_Io;         -- text IO library package
with Ada.Float_Text_IO; use Ada.Float_Text_IO;    -- Float IO package
procedure Fahr2Cels is
```

```
   Fdeg, Cdeg : Float;
begin
   New_Line;
   Put_Line( "Fahrenheit to Celsius conversion." );
   Put_Line( "Enter 0 to terminate." );
   New_Line;
   loop                                         -- start of loop
      Put( "Degrees Fahrenheit? " );        -- no line feed
      Get( Fdeg );
      New_Line;       -- input a real
      Cdeg := ( ( Fdeg - 32.0 ) * 5.0 ) / 9.0;
      Put( "Degrees Celsius = " );
      Put( Cdeg, Aft => 2, Exp => 0 );      -- output a real
      New_Line;
      New_Line;
      exit when Fdeg = 0.0;                 -- loop exit
   end loop;
   Put_Line( "Done. Thanks for visiting." );
end;
```

In this example, we have put the exit condition right at the end of the loop. When programming loops, take care to ensure that the loop has an exit condition.

Here is a sample output:

```
     Fahrenheit to Celsius conversion.
     Enter 0 to terminate.

     Degrees Fahrenheit? 15

     Degrees Celsius = -9.44

     Degrees Fahrenheit? 95

     Degrees Celsius = 35.00

     Degrees Fahrenheit? 0

     Degrees Celsius = -17.78

     Done. Thanks for visiting.
```

Now let’s look at a few other loop structures.

2.3.2 A While loop

A While loop tests for the loop termination condition at the start of the loop:

```
with text_io; use text_io;
with ada.integer_text_io; use ada.integer_text_io;

PROCEDURE WhileLoop IS
   Counter : Integer;
BEGIN
   Put_line( "Set up a while loop to print a table of squares" );
   Counter := 1;
   WHILE Counter <= 10 LOOP
      Put( counter ); put ( counter * counter );
      New_line;
      Counter := Counter + 1;
   END LOOP;
END;
```

2.3.3 A For loop

The For loop steps through a sequence of items, which may be and discrete type, such as integers, enumeration types, characters, etc. All the items must be of the same type.

```
with text_io; use text_io;
with ada.integer_text_io; use ada.integer_text_io;

PROCEDURE ForLoop IS
BEGIN
   Put_line( "Set up a for loop to print a table of squares" );
   FOR Counter in 1..10 LOOP
      Put( counter ); put ( counter * counter );
      New_line;
   END LOOP;
END;
```

For integer counting, the FOR loop is the preferred form. The variable Counter does not need to be declared. It is set up automatically and is valid only while the code is inside the loop. It is also treated as a constant within the loop, so the program cannot mess about with the number of times the loop executes.

You can also count backwards. With the While, start at 10, decrease the counter by 1 each time through the loop, and make the while test if the Counter >= 1. In the For loop, use `FOR Counter in Reverse 1..10 LOOP.` While loops, on the other hand, can use any variable in the loop, integers or reals and thus offer more flexibility.

Notice the use of ‘IN’. Ada frequently uses this construct, which is a membership form and can always be used to test if a variable is within a range or belongs to a set.

2.3.4 Loops can use any discrete type for counting, such as characters:

```
      with text_io; use text_io;
      PROCEDURE CharLoop IS

         Sp : Character := ' ';
         z : Character := 'z';
      BEGIN
         put_line( "Use a For loop to write the alphabet:" );
         FOR Ch IN Sp .. z LOOP
            Put( Ch );
         END LOOP;
      END;
```

Let’s write a simple children’s educational game as an example. Ask the player to type a letter and then ask what is the following letter in the alphabet. End when the player enters ‘z’, which has no following letter.

The logical structure of the program is a little complex, to we should start by thinking about the design of the program. To help me with this, I like to use pseudocode, which is a kind of program description that sets out the program's algorithm (recipe, or sequence of steps) in a simplified way and is a good way to think about the design of a program. It's a bit like a program in an English-like language and shows the logical structure of the program without the detail and constraints of an actual programming language. It is useful for program design in any language.

```
Get a letter and check it's a..z
Exit if it's z
If it is a..z
   Get another letter
   If it is a..z
      if it is the next letter in the alphabet
        congratulate the player
      otherwise
        say no, that's wrong
   If the letters entered were not a..z
     Tell the player to only enter a..z
End the loop
```

If you check the code, you will see it follows the pseudocode closely.

Here is the code:

```
WITH Text_Io; USE Text_Io;
PROCEDURE Alphabet IS
   Ch1, Ch2 : Character;
BEGIN
   Put_Line("Type a letter (a..z). Typing z ends the game");
   LOOP
      New_Line; Put( "Type a letter: " );
      Get_Immediate(Ch1); Put(Ch1);
      New_Line(2);
      Ch2 := ' ';   -- in case we don't read a new Ch2
      IF Ch1 IN 'a'..'z' THEN
         BEGIN
            EXIT WHEN Ch1 = 'z';
            Put("What is the next letter in the alphabet? ");
            Get_Immediate(Ch2); Put(Ch2);
            New_Line;
         END;
      END IF;
      IF Ch2 IN 'a'..'z' THEN
         BEGIN
            New_Line;
            IF Ch2 = Character'Succ(Ch1) THEN
               Put_Line("That's right, well done");
            ELSE
               BEGIN
                  Put("Sorry, that's wrong. The next letter is ");
                  Put(Character'Succ(Ch1));
                  New_Line;
               END;
            END IF;
         END;
      END IF;
      IF (Ch1 NOT IN 'a'..'z') OR (Ch2 NOT IN 'a'..'z') THEN
         Put_Line("You must type a letter 'a' to 'z'");
      END IF;
   END LOOP;
   New_Line; Put_Line("Thanks for playing");
END;
```

In many other languages, there is a third scheme, the Repeat … Until loop. Ada does not offer this. Instead, you can just put an exit statement at the end of the loop, as we have seen in the temperature conversion program above, which exits when a Fahrenheit temperature of zero is entered. This form achieves the same behaviour as the repeat … until:

```
LOOP
   <statements>
   Exit when <condition>;
END LOOP;
```

2.3.5 Fibonacci numbers

Here is a bigger example, using loops, ifs and tests. It is a program to calculate Fibonacci numbers. Fibonacci numbers are used in lots of maths puzzles and associated games. A quick web search for Fibonacci puzzles will open a world of fascinating puzzles for you. They are also used in some graphics rendering and encryption systems. For now, we will simply show how to calculate Fibonacci numbers with a program loop.

Fibonacci numbers are defined as follows:

F(0) = 0, F(1) = 1, F(N) = F(N-1) + F(N-2)

To see how this works, let's calculate a few numbers in the sequence:

F(2) = F(0) + F(1) = 0 + 1 = 1
F(3) = F(1) + F(2) = 1 + 1 = 2
F(4) = F(2) + F(3) = 1 + 2 = 3
F(5) = F(3) + F(4) = 2 + 5 = 5

The sequence continues as 8, 13, 21, 34 etc. As you can see, it starts to grow increasingly rapidly.

So, to calculate F(N), we start a loop from 2, with starting values of 0 and 1, then run the loop to N, continuously adding the two previous results together to form the next result. We will define variables Fnew, F and Fold, and then calculate Fnew := F + Fold, then make Fold := F and F := Fnew. Starting with F := 1 and Fold = 0, the sequence will run as follows

Fnew	F	Fold
-	1	0
1	1	1
2	1	1
3	2	1
5	3	2

Etc.

Now we know what we want to calculate, here is the pseudocode.
This program

a) Gives a startup message
b) Starts the main loop

c) Asks the user if they want to continue and exits if not
d) Inputs the value N
e) Calculates F(N) = 0 1 1 2 3 5 . . where F(n) = F(n-1) + F(n-2), F(0) = 0, F(1) = 1
f) Displays the result
g) Repeats the main loop

Here is the code, which follows the pseudocode closely:

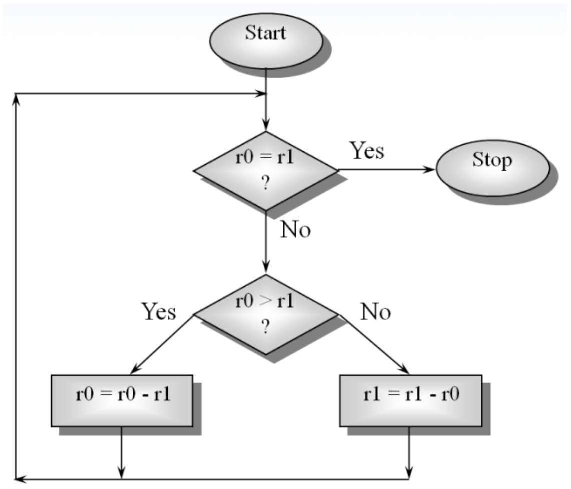
```
WITH Text_Io; USE Text_Io;
WITH Ada.Integer_Text_IO; USE Ada.Integer_Text_IO;

PROCEDURE FibN IS
   Fold, F, Fnew, N : Integer;
   Ch   : Character := 'n';
BEGIN
   Put_Line( "Displays fibonacci numbers until user quits" );
   LOOP
      New_Line;
      Put( "Want another Fibonacci number (answer y for yes): " );
      Get( Ch );
      EXIT WHEN (Ch /= 'y') AND (Ch /= 'Y');  -- accept small or capital y
      Put("Enter a number >= 1 for which you want the Fibonacci number: ");
      Get(N);
      IF N >= 1 THEN
         BEGIN
            Fold := 0;
            F := 1;
            FOR I IN 2..N LOOP
               Fnew := F + Fold;
               Fold := F;
               F := Fnew;
            END LOOP;
         END;
      END IF;
      Put( "Fibonacci number " );
      Put( N, Width => 1 );
      Put( " = " );
      Put( F, Width => 1 );
      New_Line;
   END LOOP;
END;
```

This program embeds multiple structures within each other:
Outer loop
  Loop exit test
    If statement
      Inner loop inside the If statement

It shows how Ada can flexibly embed structures within each other.

2.3.6 Calculating the greatest common divisor (GCD) of two numbers

Here is a further example of the use of an IF statement and a loop. The program calculates the greatest common divisor (GCD) of two given integers. The GCD is the biggest integer that can divide the two given integers with no remainder. The classic algorithm is described by this flow chart. Flow charts are particularly useful for visualising choices.



The pseudo-code might look like this:

```
Read in two numbers, r0 and r1
If either are zero, exit
Loop until the numbers are equal
  If the numbers are equal, r0 is the result
  If r0 > r1, r0 := r0 – r1;
  Otherwise r1 := r1 - r0;
End loop
```

We can now translate this to Ada:

```
WITH Text_Io; USE Text_Io;
WITH Ada.Integer_Text_IO; USE Ada.Integer_Text_Io;
PROCEDURE GCD IS
   R0, R1 : Integer;
BEGIN
   Put_line("Calculate GCD of two numbers.");
   OuterLoop: LOOP
      Put("Enter first Number: "); Get(R0);
      EXIT OuterLoop WHEN R0 = 0;
      Put("Enter second number: "); Get(R1);
      EXIT OuterLoop WHEN R1 = 0;
      New_Line;
      InnerLoop: LOOP
         EXIT InnerLoop WHEN R0 = R1;
         IF R0 > R1 THEN R0 := R0 - R1;
         ELSE R1 := R1 - R0;
         END IF;
      END LOOP InnerLoop;
      Put( "GCD is: "); Put(R1);
      New_Line;
   END LOOP OuterLoop;
END;
```

In this example, because there are two loops and three exit statements, we have added labels, InnerLoop and OuterLoop, to the loops, so we can be absolutely sure which loops the exit statements apply to. Since unlabelled exit statements exit the loop in which they are found, the labels are not strictly necessary, but in complex programs they can make the structure much clearer.

We have also used a simple IF .. ELSE .. construct to implement the algorithm, and this neatly shows the power of this kind of IF statement. With this example, you have the core for the GCD game – google GCD game and have fun.

2.3.7 Leap year calculations

Here is another example of the use of IF statements, in the calculation of leap years. A year is a leap year if it is exactly divisible by 4, not a leap year if exactly divisible by 100, but is a leap year if it is exactly divisible by 400. We can code this as follows:

```
with text_io; use text_io;
WITH Ada.Integer_Text_IO; USE Ada.Integer_Text_Io;

PROCEDURE LeapYears IS
   Leap : Boolean;
   Year : Integer;
BEGIN
   put_line( "Leap Year checker" ); new_line;
   LOOP
      Put( "Enter year (0 to quit) .. " );
      get( year );
      exit when year = 0;
      IF Year MOD 100 = 0
         THEN Leap := ( Year MOD 400 ) = 0;
         ELSE Leap := ( Year MOD   4 ) = 0;
      END IF;
      IF Leap
         THEN Put( Year ); Put_line( " is a leap year" );
         ELSE Put( Year ); Put_line( " is not a leap year" );
      END IF;
   END LOOP;
END;
```

Notice we define the variable LeapYear as type Boolean, so it can only take the values True or False. The MOD function calculates the remainder of an integer division, if zero the division is exact. So the expression (Year MOD 400) = 0 evaluates to True if the division is exact (i.e. has zero remainder).

2.3.8 The Game Loop

In computing, those parts of programs that repeat are called loops. All games are built around loops, which describe the flow of events in a game. When you design a game, the idea is to break up the game into parts, called components, that repeat inside the game loop.

The game loop is usually made up of 6 parts: startup, which loads the initial settings and give the player an introduction to the game story and what the user has to do; inputs from the player; updating the game state such as the player’s position and score by applying the game rules to the player input; updating the display, both text and graphics; checking if the game is over, e.g. if the player is dead, has run out of time and going back to get more player input if the game is not over. If the game is over, the game shuts down, the player is given the final score and the program saves information such as game levels and scores, then terminates.

If you look closely at the previous examples in this section (2.3) you will see that they follow the basic structure of the game loop.

Out final example in this section is the classic game of Guess my Number. Before getting into it, however, we need to take a small detour and talk about random numbers. Random numbers add a whole level of excitement to a game by making it unpredictable and adding an element of surprise. All casino games include randomness, through die rolls, card shuffles, wheel spins, etc.

Computer programs need to include functions to generate random numbers to program these effects. Here is a simple program that uses random numbers

2.3.9 The Die Roller Program

This program simulates the roll of a six-sided die. The computer does this by generating a random number in the range 1 to 6. Here is the code:

```
-- die roll simulation

WITH Text_IO; USE Text_IO;
with ada.Integer_Text_IO; USE ada.Integer_Text_IO;
with Ada.Numerics.Discrete_Random;

procedure DieRoll is
   subtype DieT is Integer range 1 .. 6;
   package Random_Die is new Ada.Numerics.Discrete_Random (DieT);
   use Random_Die;
   G : Generator;
   Die : DieT;
begin
   Reset (G);  -- Start the generator in a unique state in each run
   -- Roll a die
   Die := Random(G);
   put("You rolled a "); put(Die, 1); new_line;
end DieRoll;
```

The program starts with the usual naming of the packages to be used, namely Text_IO, Integer_Text_IO and then a new package, Discrete_Random, which is part of the larger Numerics package. We then define a subtype for the die, called DieT which can take values 1 to 6, so we give it that range. Next, we have to make version of the random number package for this subtype. The Discrete_Random package is actually a generic package, and this creates a specific instance of it. The package provides a procedure, Reset(G) which randomises, or seeds, the number sequence, and a function, Random(G), which will return a random number in the range 1 to 6 as specified for the type DieT. The random package requires that a variable of type Generator, which we call G here, is used internally by the package. So we call Reset(G) to begin, then call Random(G) to get a random number to store in the variable Die, which it then displays.

Functions in general are pieces of code that can do some work and return a value. You call or invoke a function by using its name. You can pass information to the function through one or more arguments within a pair of parentheses. If a function returns a value, you can assign that value to a variable. That's what we do here with the assignment statement Die := Random(G);. Here, the value returned by the function Random gets assigned to the variable Die. The function requires the argument G to be passed to it. G, in this case, is a variable of the type Generator that is built in to the random number package that it needs for its work.

The Ada random function generates a random number in the range specified by the type, which here is DieT, which we gave a range of 1..6. When we make an instance of the package, we can specify any legal Ada discrete subtype, like a range of integers, characters or enumeration types. Functions can also take arguments, which are values to use in their work. You provide these values by placing them between parentheses after the function name, separated by commas. These supply values that are passed to the function. Another example is the square root math function, for example y := sqrt(x). There, the value of x is passed to the function, which returns a result, the square root of x, which is then store in the variable y.

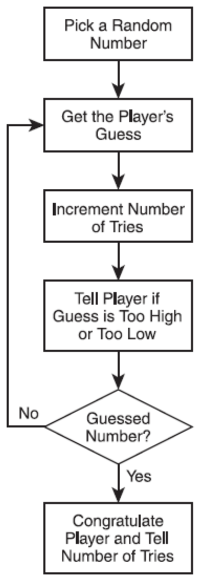
Seeding the Random Number Generator

Computers generate pseudorandom numbers—not truly random numbers— based on a formula. One way to think about this is to imagine that the computer reads from a huge book of predetermined numbers. By reading from this book, the computer can appear to produce a sequence of random numbers. But there's a problem: The computer always starts reading the book from the beginning. Because of this, the computer will always produce the same series of "random" numbers in a program. In games, this isn't something we'd want. We wouldn't, for example, want the same series of dice rolls in a game of craps every time we played. A solution to this problem is to tell the computer to start reading from some arbitrary place in the book when a game program begins. This process is called seeding the random number generator. Game programmers always seed the random number generator so the random sequence starts in a new place every time the program is run.

In Ada, the procedure call Reset(G), provided by the discrete random package, randomises the generator. This is called a procedure, rather than a function, since it does not return a value that can be assigned to another variable. Many languages do not distinguish between procedures and functions and only have functions, some of which just don't return assignable values.

2.3.10 Number guessing game

In this classic number guessing game, the computer selects a random number from 1 to 100, and the player tries to guess the number in as few moves as possible. Each time the player enters a guess, the computer tells him whether the guess is too high, too low, or right on the money. Once the player guesses the number, the game is over. Here is the game design:


Pick a Random Number
Get the Player's Guess
Increment Number of Tries
Tell Player if Guess is Too High or Too Low
No
Guessed Number?
Yes
Congratulate Player and Tell Number of Tries

In pseudocode form, we can write this as

```
        Set up the game
Loop:   get player input
        Update the guess
        Update the display
        If game not over, loop
        Shut down
```

Here is the program:

```
-- guess secret number in range 1 to 100

WITH Text_IO; USE Text_IO;
WITH Ada.Integer_Text_IO; USE Ada.Integer_Text_IO;
WITH Ada.Numerics.Discrete_Random;
```

```
PROCEDURE GuessMyNumber IS
   Tries : Integer := 0;
   Guess : Integer;
   SUBTYPE NumT IS Integer RANGE 1 .. 100;
   PACKAGE RandomNum IS NEW Ada.Numerics.Discrete_Random (NumT);
   USE RandomNum;
   G         : Generator;
   SecretNum : NumT;

BEGIN
   Reset(G);
   SecretNum := Random(G); -- returns random number between 1 and 100
   Put_Line("Welcome to Guess My Number"); New_Line;
   LOOP
      Put( "Enter a guess: ");
      Get(Guess);
      Tries := Tries + 1;
      IF (Guess > SecretNum) THEN
         BEGIN
            Put_Line("Too high!"); New_Line;
         END;
      ELSIF (Guess < SecretNum) THEN
         BEGIN
            Put_Line("Too low!"); New_Line;
         END;
      ELSE
         BEGIN
            New_Line; Put("That's it! You got it in ");
            Put(Tries, 1); Put_Line(" guesses!");
            EXIT; -- exits the loop when Guess = SecretNum
         END;
      END IF;
   END LOOP;
END;
```

Exercise: add a condition to limit the number of tries to 7 and gives the message “Too many tries, you lose” is the players uses more than 7 tries. (Using the halving method, often called binary division, it is always possible to guess the number in 7 tries or less.)

Now let’s try a number arithmetic game, very often used to teach children basic arithmetic.

The pseudocode is simple:

```
Generate two numbers and display them
Give the player 3 tries to get the right answer
Ask for the answer
If the answer in correct, say so and increment numRight
Otherwise say it’s wrong and increment numWrong
Loop until player enters a zero
Display numRight and numWrong
```

```
WITH Text_Io;
USE Text_Io;
WITH Ada.Integer_Text_IO;
USE Ada.Integer_Text_IO;
WITH Ada.Numerics.Discrete_Random;
```

```
PROCEDURE AddTwo IS
   SUBTYPE Vals IS Integer RANGE 10..99;
   PACKAGE Random_Val IS NEW Ada.Numerics.Discrete_Random (Vals);
   USE Random_Val;
   G : Generator;
   First, Second, Ans, NumRight, NumWrong : Integer   := 0;
BEGIN
   Reset(G);
   Put_Line("You have 3 tries to add 2 numbers.");
   Put_Line("Type in the answer at the prompt.");
   Put_Line("Type in 0 to end.");
   Mainloop: LOOP
      New_Line;
      First := Random(G); Second := Random(G);
      Put(First, 2); Put(" + "); Put(Second, 2); Put(" = " );
      FOR I IN 1..3 LOOP
         Get(Ans);
         Skip_Line;
         EXIT MainLoop WHEN Ans = 0;
         IF Ans = First + Second THEN
            Put_Line("Correct");
            Numright := Numright + 1;
            EXIT; -- inner loop
         ELSE
            Put("Sorry,that's wrong, try again. ");
            NumWrong := NumWrong + 1;
            IF I = 3 THEN
               Put("The correct answer is ");
               Put(First + Second, 1);
               New_Line;
            END IF;
         END IF;
         New_Line;
      END LOOP;
   END LOOP MainLoop;
   Put("You got "); Put(Numright, 1); Put_Line(" correct");
   Put(" and "); Put(NumWrong, 1); Put_Line( " wrong.");
END;
```

Exercises: Add levels to the game so the number ranges vary (eg 1 : 1..10, 2 : 10..50, etc.) and then add subtract and multiply. Next add division, but this is trickier as you might want to choose only numbers that result in whole number division (ie no remainder).

2.3.11 Javelin flight

Now that we understand loops and strings, let's extend the javelin throw program to show the flight of the javelin throw (its trajectory) on the screen. We haven't covered graphics libraries, so will just use text IO to display the throw in character form, with the distance shown vertically on the screen and the height of the javelin horizontally.

To do this, we need another formula for the height of the javelin in flight. From Wikipedia, we get the formula h = v.t.sin(45) – t^2G/2. In the program v is the initial velocity, t is the time into the flight and sin(45) = 0.7071, for a throw at a 45° angle. For t, we will calculate a step size for the plot based on the distance thrown and plot 0 .. integer(distance) steps, with each step equal to the flight time divided by the flight distance in meters. The program now looks like this:

```
      --  ADA floats example - throwing a javelin
```

```
WITH Text_Io; USE Text_Io;
WITH Calendar; USE Calendar;
WITH Ada.Float_Text_IO; USE Ada.Float_Text_IO;

PROCEDURE Javelin IS
   G           : CONSTANT := 9.81; -- Acceleration due to gravity
   Stop, Start : Time;
   Taken       : Duration;
   Char        : Character;
   Length      : Integer;
   Runup       : String (1 .. 20);
   Velocity, Distance, FlightTime, TimeStep, ThisStep : Float;
BEGIN
   Put_Line("hit any key 20 times to time your runup");
   Get_Immediate(Char); -- gets one character immediately it is
struck
   Start := Clock;
   Get_Line(Runup, Length);
   Stop := Clock;
   Taken := Stop - Start; -- runup time
   New_Line;
   IF (Taken > 10.0) THEN
      Put_Line("This is not a valid throw - too slow");
   ELSIF (Length < 20) THEN
      Put_Line("This is not a valid throw - short runup");
   ELSE
      Put("You took ");
      Put(Float(Taken), Fore => 1, Aft => 2, Exp => 0);
      Put_Line(" seconds for your runup");
      Put("You threw the javelin at ");
      Velocity := (10.0 - Float(Taken)) * 2.0;  -- set throw velocity
                  -- using any formula you like based on runup time
      Put(Velocity, Fore => 1, Aft => 2, Exp => 0);
      Put_Line(" meters per second");
      Distance := (Velocity ** 2) / G;  -- from physics, based on
      FlightTime := 2.0*Velocity*0.7071/G;  -- throw at 45 degrees
      Put_Line("You threw it ");
      Put(Distance, Fore => 1, Aft => 2, Exp => 0);
      Put_Line(" meters.");
      Put_Line("It flew for ");
      Put(FlightTime, Fore => 1, Aft => 2, Exp => 0);
      Put_Line(" seconds.");

-- now draw the curve for the throw
      Put_Line("Its flight (plotted vertically) is:");
      Put_Line("x");
      TimeStep := FlightTime / Distance;
      FOR I IN 1..Integer(Distance) LOOP
         ThisStep := TimeStep * Float(I);
         FOR J IN 1..Integer(5.0*(Velocity*ThisStep*0.7071-
                                 (ThisStep**2)*G/2.0)) LOOP
            Put(" ");     -- 5.0 above sets vertical scale
         END LOOP;
         Put_Line("x"); -- marks position of javelin
      END LOOP;
   END IF;
END;
```

The output, showing the curve of the throw, looks like this (next page)

```
Hit any key 20 times to time your runup
ddddddddddddddddddddd

You took 1.55 seconds for your runup
You threw the javelin at 16.89 meters per second
You threw it 29.09 meters.
It flew for 2.44 seconds.
Its flight (plotted vertically) is:
x
     x
         x
             x
                 x
                     x
                        x
                           x
                             x
                               x
                                 x
                                  x
                                   x
                                    x
                                    x
                                    x
                                    x
                                   x
                                  x
                                 x
                               x
                             x
                           x
                        x
                     x
                  x
              x
          x
     x
x
```

Note we have also added code to check the run-up is not short or too slow.

3. Programming errors

3.1 Compile time errors

We saw in section 1.3 above how errors in entering the code can prevent the program from being compiled. This results in error messages when we try to compile the program, which must be corrected before we can proceed. These are called compile time errors. Other errors, for example calling a non-existent library package, might result when the compiled program is built into a runnable unit, also called an executable. Both of these kinds of error will show up in AdaGIDE, or in any most development environments, with error messages that help you to correct the problem.

3.2 Run time errors

Other errors can occur when the program is run. For example, a number may go out of its allowable range. For example, if we run the Fibonacci program in FibN.ada above, and enter too large a value for N we will get an error. Here is what happens:

```
Displays fibonacci numbers until user quits

Calculate another Fibonacci number (answer y for yes): y
Enter the number >= 1 for which you want the Fibonacci number: 40
Fibonacci number 40 = 102334155

Calculate another Fibonacci number (answer y for yes): y
Enter the number >= 1 for which you want the Fibonacci number: 50

raised CONSTRAINT_ERROR : FibN.adb:28 overflow check failed
```

The first calculation for F(40) is ok, but the second one for F(50) raises a constraint error due to an overflow check failing. This means we have tried to calculate a number that is too large for a standard integer to hold. The calculation fails and the system gives an error message, which shows the program unit FibN and the line number, 28, in the source code where the failure occurred. All integers in computers have a limited range (it may be big, but it is not infinite). On 16-bit computers, an integer has a maximum value of $2^{15} - 1$, while on 32-bit machines it is $2^{31} - 1$. GNAT allows long_long_integers that go up to $2^{63} - 1$. Try changing the program to use long_long_integers.

Floats also have a limited range and a limited precision, which means that when we compute with floats, our results may carry many small errors, which can accumulate. For example, this small program should give a result of 300 000. It actually gives a result of 2.99547E+05. This is because 0.3 cannot be represented exactly in binary, so it carries a small error, which accumulates each time the program repeats the loop. Programmers say the calculation picks up dirt, and the more it is repeated, the dirtier it gets.

```
WITH Text_Io; USE Text_Io;
WITH Ada.Float_Text_IO; USE Ada.Float_Text_IO;

PROCEDURE FltDirt IS
   Val : Float := 0.3;
   Tot : Float := 0.0;
BEGIN
   put("0.3 * 1 million = ");
   for I in 1 .. 1_000_000 loop
      Tot := Tot + Val;
   END LOOP;
```

```
      Put(Tot);
      new_line;
   end;
```

So you see that Ada checks programs at run time and tells you if it detects a problem arising from numerical overflow, attempted divide by zero, etc., but some errors, such as small numerical errors, can't be detected automatically and the programmers has to check that the result in within reasonable range of that expected.

Floating point calculations are also prone to errors when mixing large and small numbers. 32-bit (single precision) floats have a precision of 7 digits. So let's suppose for example we want to add 10000 and 0.0001. Each can be stored in a single precision float and adding them gives 10000.0001. The result, however is stored in single precision as 1.000000*10^4 (= 10000 !). However, the result is still accurate to 7 significant figures, or 1 part in 8 million and the small amount is lost. As a rule, if you need to add lots of numbers, some big and some small, sort them first in order of size, then add the small numbers first and work up to the larger numbers.

Many languages do not provide any runtime checks and allow the program to continue despite runtime errors, sometimes with catastrophic results. This type of error caused an Ariane rocket launching 4 satellites to crash in 1996, at a cost of US$370 million. See https://en.wikipedia.org/wiki/Cluster_(spacecraft) .

3.3 Logical errors

A more difficult type of error occurs when you make a mistake with your algorithm. For example, if the inner loop of the Fibonacci program said `FOR I IN 1..N LOOP` instead of the correct code, `FOR I IN 2..N LOOP` GNAT will happily compile and run the program, but give you incorrect results (it would calculate the sequence 0 1 2 3 5 8 etc. instead of 0 1 1 2 3 5 etc.).

So the situation here is worse than the old saying, Garbage In, Garbage Out. With this kind of error, even with correct data in, we would get garbage out

To deal with this kind of problem, we first require careful design of the program, then careful checking and testing of the program. In this example we would start by testing that we got the correct answer for N=1 and N=2 and we would see right away there was an error. The key is to test each new piece of code you write. Code a little, but test a lot!

Both AdaGIDE and the GNAT programming environment offer good debugging facilities, where the user can run a program up to a point in the code, called a breakpoint, which can be set to any line in the program. When the program reaches that point, it stops and the user can check the values of variables, step the program along line by line and generally monitor its execution.

Some logical errors are very subtle and can be hard to find. As we will see later, Ada2012 offers some ways to try to prevent these kinds of errors from slipping past us. Good programmers pay attention to compiler warnings!

3.4 Exception handling

If a runtime error occurs and it raises an exception, is there anything we can do about it?
In fact, there is. Ada allows us to catch and handle exceptions and even to define our own exceptions.

Exceptions give you a general way to handle runtime errors, including user input errors, using exception handlers to specify remedial actions or proper shutdown in case of errors.

Any begin-end block can have an exception handler:

```
     x, y : integer;
     -- other declarations
  Begin
     -- Get values for x, y
     x := x / y;
  exception
     when Constraint_Error =>   -- result out of range
        Put_Line (“Possible division by zero, re-enter x, y values”);
     when others  => Put_Line (“Ooops, unexpected error!”);
  end;
```

The ‘others’ keyword indicates all other possible exceptions.

Sometimes exceptions are not used for programming errors, but for checks. A common use for exception handlers is to check that inputs are reasonable and of the correct type. It is always wise to do this in programs because users often enter wrong data. The program below catches two exceptions, a possible input error and a calculation overflow

```
WITH Text_Io; USE Text_Io;
WITH Ada.Integer_Text_IO; USE Ada.Integer_Text_Io;
PROCEDURE ProgWithExceptions IS
   N : Integer;
BEGIN
   Put_Line("Calculate a table of squares, cubes and powers of 4.");
   LOOP          --   as long as input is not legal
      BEGIN
         Put("Table range 1..N. Enter N: ");
         Get (N);  -- a non-integer or bad value will cause an exception
         EXIT;     --  if got here, input is valid, so exit from the loop
      EXCEPTION
         WHEN OTHERS =>
            Put_Line ("Input must be an integer, try again");
            Skip_Line;  --  clear the input buffer
      END;
   END LOOP;
   New_Line; Put_Line("We have a good value for N, so print the table");
   Put_Line("          N        N^2        N^3        N^4");
   FOR I IN 1 .. N  LOOP
      BEGIN
         Put (I); Put(I*I);
         Put(I**3);  -- raise to the power of 3
         Put(I**4);  -- and 4
         New_Line;
      EXCEPTION
         WHEN OTHERS =>
            New_Line; Put_Line("That's all I can do");
            EXIT;
      END;
   END LOOP;
   Put_Line("Bye now. Thanks for visiting.");
END;
```

The input loop will reject a non-integer input, the second loop will terminate when the power of 4 calculation overflows.

Now let’s take another look at the GuessMyNumber game. The user’s guess must be in the range 1..100. The input code was:

```
      Put( "Enter a guess: ");
      Get(Guess);
```

We can check that the guess is in range by testing the input with an IF statement:

```
      loop
         Put( "Enter a guess: ");
         Get(Guess);
         IF Guess IN 1..100 THEN
            EXIT;
         ELSE
            Put_Line("That's not in the range 1 to 100");
         END IF;
      END LOOP;
```

Using Ada exceptions, we can check the input by defining guess as having a range 1..100 which will result in an exception if the user enters a number outside of that range. This has the further advantage that it also catches any other input error, such as entering something that is not an integer:

```
      LOOP
         begin
           Put( "Enter a guess: ");
            Get(Guess);
            exit;
           Exception
              when others =>
                Put_Line("That's not a number in the range 1 to 100");
         end;
      END LOOP;
```

This use of exceptions is a common idiom in Ada.

Exceptions are often used with files to catch conditions like a wrong file name or trying to read past the end of a file. We will use this later when we introduce file handling.

4. More types

4.1 Enumeration types

Ada supports lots of types and data structures, including arrays, records and enumeration types. We will start by looking at enumeration types. We have already met them, briefly. They are lists of items with specific names or values listed. For example

```
      type Primary_Colours is (red, blue, yellow);
      type Traffic_Light is (green, amber, red);
      type Week_Day is (Mon, Tues, Weds, Thurs, Fri, Sat, Sun);
```

We can then declare variables of these types, and optionally initialise them:

```
      myColours : Primary_colours := yellow;
      NS, EW : Traffic_Light := red;
      Working_Day : week_day;
```

In traditional languages, information like this is represented numerically and coded with specific values. For example, traffic light colours might be represented numerically as 1, 2 and 3 for green, amber and red. Enumeration types allow us to work directly with a representation that is much closer to the real world. This makes the intention of the programmer clearer. For example:

```
      NS := green;
      IF (NS = green) and (EW = green) then danger;
```

To do IO with an enumeration type, we must make an instance of the IO package for that type. We will discuss packages and instances soon.

Here is an example using an enumeration type.

```
WITH Text_Io; USE Text_Io;

PROCEDURE UsingSpices IS
   TYPE Spices IS (Basil, Cayenne, Coriander, Rosemary);
   PACKAGE Spice_Io IS NEW Enumeration_Io(Spices); USE Spice_Io;
BEGIN
   Put_Line("My spices are:");
   FOR I IN Spices'RANGE LOOP    -- I is of type spices and takes the
      Put(I); Put("  ");         -- values Basil, Coriander, etc.
   END LOOP;
   new_line(2);
   put_line("Here are some uses for my spices.");
   FOR I IN Spices'RANGE LOOP  -- use the enumeration type in a loop
      put(I); put (" is ");
      CASE I IS
         WHEN Basil => Put_Line("great in pesto");
         WHEN Cayenne => Put_Line("hot and spicy, works well with meat");
         WHEN Coriander => Put_Line("excellent in soups");
         WHEN Rosemary => Put_Line("a subtle taste in sauces");
      END CASE;
   END LOOP;
END;
```

The output from the program looks like this:

```
My spices are:
BASIL  CAYENNE  CORIANDER  ROSEMARY

Here are some uses for my spices.
```

```
BASIL is great in pesto
CAYENNE is hot and spicy, works well with meat
CORIANDER is excellent in soups
ROSEMARY is a subtle taste in sauces
```

Some points you should note:

```
   TYPE Spices IS (Basil, Cayenne, coriander, rosemary);
```

This line defines the enumeration type Spices

```
   PACKAGE Spice_Io IS NEW Enumeration_Io(Spices); USE Spice_Io;
```

This makes a new instance of the enumeration IO package for our new type. All enumeration IO, however, is in capitals.

```
   FOR I IN Spices'RANGE LOOP
```

This makes a loop over the range of spices from the first, Basil, to the last, Rosemary. The variable I in the loop is of type Spices and can be used in IF or Case statements, as the program shows

The **RANGE** in **Spices'RANGE** is called an Attribute**.** All types and their associated variables have attributes, which are the characteristics of the type, such as their minimum and maximum values, their range, etc. Attributes are widely used in Ada. Attributes allow you to get —and sometimes set— information about objects or other language entities such as types. A good example is the Size attribute. It describes the size of an object or a type in bits. The Language Reference Manual discusses attributes in section 4.1.4 and gives a list of all attributes of all the types in section K.2: Language-Defined Attributes

Some of the attributes of type Spices are

```
Spices’FIRST -- Basil
Spices’LAST  -- Rosemary
Spices’RANGE -- Basil .. Rosemary
X : spices; -- make a variable of type Spices
X := Rosemary; -- assign a value to the variable X
Spices’POS(X); -- the position in the list of X.
Spices’Succ(X);  -- the next item (successor) in the list
Spices’Pred(X);  -- the previous item (predecessor) in the list
Spices’Val(N);   -- the item in the list at position N
Spices’IMAGE(X); -- the string representing X.

Here are some more attributes for other types we have already met:
Integer’FIRST -- is the smallest integer on a given system
Integer'LAST  -- is the largest integer on a given system
Integer’IMAGE(N)  -- the text string representing N
FLOAT'DIGITS  -- number of significant figures
FLOAT'FIRST   -- minimum on given system
FLOAT'LAST    -- maximum on given system
FLOAT'SMALL   -- smallest on given system
CHARACTER'PRED(char) -- PREDecessor, e.g. CHARACTER'PRED('N') = 'M'
CHARACTER'SUCC(char) -- SUCCessor, e.g. CHARACTER'SUCC('N') = 'O'
CHARACTER'POS(char)  -- ASCII code, e.g. CHARACTER'POS('A') = 65
CHARACTER'VAL(int)   -- character for int, e.g. CHARACTER'VAL(66) = 'B'
```

Enumeration types are lists of items. Similarly strings are lists of characters. Both are kinds of arrays.

4.2 Arrays

Arrays can be lists, or tables, or blocks of data of 1, 2 or more dimensions. A list has one dimension, so has one index, while a table has two dimensions, so has two indices. We can make arrays with 3, 4 or more dimensions, accessed using their indices.

As mentioned above, strings are a kind of array, in that a string is a list of characters, with the individual characters accessed via the index, which runs from 1 to the length of the string. (This is different from C, where the string index runs from 0 to the length-1).

So in the string myName : string := "Dave Levy" the length is 9, the character index runs from 1 to 9, myName(1) = 'D' and myName(6) = 'L'.

We can make lists of any type, with the restriction that all items in the list must be of the same type, such as characters, integers or floats. For example:

```
type marks is array (1..max_size) of FLOAT;  -- defines a type of array
myClass : marks ;  -- make an instance of the array for my class.
```

Notice the array has a specific size and how the definitions are similar to strings. In fact, a simple string in Ada is just a predefined array type of characters.

The item in brackets is called the index and can be of any range or discrete type – integer, character or enumeration, and the array elements can be any legal type. For example:

```
Type year_of_birth is array (1900 .. 2020) of integer;
```

```
Type Names is array(1 .. max_size) of string(20);
```

Now we can fill the list with marks and do things like calculate average marks. For example, to initialize the data in myClass, we could write:

```
myClass : marks := (9.3, 8.2, 5.7, others => 0.0);
```

This puts values in the first 3 items in the list and sets all the rest to 0.0.

We can then do calculations with the array elements:

```
totalSoFar := myClass(1) + myClass(2) + myClass(3); -- add some elements
for I in marks'RANGE Loop     -- loops from marks(1) to marks(max_size)
   IF myClass(I) = 0.0 then   -- tests a element
      Put_line("Bad student!");
   END IF;
END LOOP;
```

Here is an example to calculate the average of a set of non-zero marks.

```
WITH Text_Io; USE Text_Io;
WITH Ada.Float_Text_IO; USE Ada.Float_Text_IO;

PROCEDURE MarksAvg IS
   Class_Size : CONSTANT integer := 10;
   Total : Float := 0.0;
   numNotZero : integer := 0;
   TYPE Marks IS ARRAY (1 .. Class_Size) of float; -- defines an array type
   MyClass : Marks := (3.5, 0.0, 8.7, 6.9, 9.1, OTHERS => 0.0); -- make an
      -- instance of the array for my class and load it with some marks
BEGIN
   Put_Line("Class marks are:");
   FOR I IN Marks'RANGE LOOP
      put(myClass(I), Fore => 2, Aft => 1, Exp => 0);
      Total := Total + myClass(I);
      If myClass(I) /= 0.0 then numNotZero := numNotZero + 1; end if;
      Put("  ");
   END LOOP;
   New_Line(2);
   Put("Average of non-zero marks is ");
   IF NumNotZero /= 0 THEN
      Put(Total / float(NumNotZero), Fore => 2, Aft => 2, Exp => 0);
   END IF;
END;
```

As you can see, the index (I) to the list myClass(I) is a simple integer. In most cases, integers are used for the index, and in many languages, only integers can be used. Ada, on the other hand, allows you to use any discrete type (integers, characters or enumeration types) with any range for an index.

Lists are very powerful and are much used in programming. Tables are even more powerful and can be set up and used much like a spreadsheet. They are two-dimensional arrays and so have two indices. In this example, we wish to keep track of a set of class marks for several students doing several marked tasks. We will use enumeration types for the students' names and for the work items, and initialise the marks table as follows:

```
Type students is (Barry, Peter, Mary, Joe);
Type work is (quiz1, lab, quiz2, total);
allMarks is array(students, work) of integer;
myClass : allmarks :=
  -- quiz1  lab  quiz2  total
   ((  20,   18,    15,    0),  -- Barry's marks
    (  17,   14,     9,    0),  -- Peter's marks
    (  19,   18,    17,    0),  -- Mary's marks
    (  20,   20,    19,    0))  -- Joe's marks
```

The data in the table is entered by rows. In this example, the totals are still to be calculated. We can now calculate the total for each student with two loops. For each student, we total the marks over the columns quiz1, lab, quiz2 and store the result in the total for each student:

```
    For row in students'range loop
                -- each row in the table contains one student's marks
       For col in quiz1 .. quiz2 loop
                -- in the columns quiz1, lab1, quiz2
          myClass(row, total) := myClass(row, total) + myClass(row, col);
                -- accumulate the total into the total column of the row
       end loop;
    end loop;
```

Here is a complete example for carrying out these calculations and printing the results:

```
WITH Ada.Text_Io; USE Ada.Text_Io;
WITH Ada.integer_Text_IO; USE Ada.integer_Text_IO;

PROCEDURE StudentsMarks IS
```

```
   TYPE Students IS (Barry, Peter, Mary, Joe);
   package names_IO is new Ada.Text_IO.Enumeration_IO(students); use names_io;
   TYPE Work IS (Quiz1, Lab, Quiz2, Total);
   TYPE AllMarks IS ARRAY (Students, Work) OF Integer;
   Names   : Students;
   MyClass : Allmarks :=
  -- quiz1  lab  quiz2  total
      ((20,  18,    15,    0),  -- Barry's marks
      (17,   14,     9,    0),  -- Peter's marks
      (19,   18,    17,    0),  -- Mary's marks
      (20,   20,    19,    0)); -- Joe's marks
BEGIN
   FOR Row IN Students'RANGE LOOP
      FOR Col IN Quiz1 .. Quiz2 LOOP
         MyClass(Row, Total) := MyClass(Row, Total) + MyClass(Row, Col);
      END LOOP;
   END LOOP;
   New_Line(2);
   Put_line("Totals for each student are ");
   FOR Row IN Students'RANGE LOOP
      Put(row, width => 7);
      Put(MyClass(Row, Total));
      New_Line;
   END LOOP;
END;
```

In a real system, the data would be kept in files in disk. We will see how to do that later.

Arrays are heavily used in scientific and business applications as well as in games. They can also have more than two dimensions. Three dimensional arrays are much used, particularly in scientific and engineering applications. To declare a three dimensional array, use the same form as for a two dimensional array, but just supply three indices. For example

```
  Type ThreeD is array(10..20, 5..11, 100..115) of weekday;
```

Sometimes we want the array dimensions to be variable, so we could build a system to handle different size groups and different numbers of work items. So far, we have used only fixed size arrays, but Ada allows us to define arrays of variable size.

4.3 Variable Size Arrays

Using only arrays of fixed size is very inflexible, and is regarded as poor programming practice, so we generally prefer to leave the array size unconstrained, to be declared later.

```
type TABLE is array (POSITIVE range <>,   POSITIVE range <>) of INTEGER;

   myTable   :  TABLE(1..4, 1..6);      -- declare a 4 by 6 table
   yourTable : TABLE(1..10, 1..20);    -- declare a 10 by 20 table
```

A string in Ada is similar to an unconstrained array of type character, leaving the length to be fixed when instances are declared.

4.4 Records

Arrays are useful when all the items in the array are of the same type, but we often want to keep track of data records made up of several types. A typical example is a set of employment records for the staff of a business, or membership information for a club, or players' information in a game. Suppose our game charges players to play, and perhaps also charges for in-game purchases. Here then is a player's basic member record:

```
   Type Player is
      RECORD
         Name       : String(1..30) := (others => ' ');  -- initialised
         Phone      : String(1..12) := (others => ' ');
         Charges    : Float := 0.0;
         Payments   : Float := 0.0;
      END RECORD;
```

This record is composed of four elements, two strings and two reals. Record elements can be of any valid type, such as integers, reals, strings or arrays. Similar record structures are used for many different types of applications.

Now suppose our game can have some number of players. We can make a list (array) of the records:

```
   Players : ARRAY( 1 .. NumberOfPlayers ) OF Member;
```

In the next section we will look at a program that shows the use of records in a basic membership database, but first we will look at another very powerful programming tool, procedures and functions.

4.5 Storing game resources

Arrays are often used in games to store this like a list of scores, or an inventory (a list) of players' resources. In a role playing game (RPG), a player may carry various items like weapons, magic potions, wand, etc.

This program maintains a list of a player's resources for a typical RPG. The player's inventory of resources is represented by an array. The array holds a set of enumeration types—one for each item in the player's possession. The player can trade and even find new items. Program PlayersInventory below shows the program in action. Of course, you could use strings instead of an enumeration type and we will see how to manipulate arrays of strings in the next example.

```
-- RPG player's Inventory
-- Demonstrates arrays

WITH Text_Io;
USE Text_Io;

PROCEDURE PlayersInventory IS
   TYPE ItemsT IS (Sword, Shield, Armour, Helmet, BattleAxe,
                   Healing_Potion, Dagger, Magic_Key, Orb, Wand, Empty);
   PACKAGE Items_IO IS NEW Text_IO.Enumeration_IO(ItemsT);
   USE Items_Io;
   TYPE Items_Array IS ARRAY (Positive RANGE <>) OF ItemsT;
   MAX_ITEMS : CONSTANT Integer := 6;
   MyItems    : Items_Array (1 .. MAX_ITEMS) := (Sword, Shield,
                                                 Armour, OTHERS => Empty);
   NumItems   : Integer := 3;

BEGIN
   Put_Line("You set out on your quest with these items");
   FOR I IN 1..NumItems LOOP
      Put(MyItems(I)); New_Line;
   END LOOP;
   Put_line("You encounter a warrior and trade your sword for a battle
axe.");
   MyItems(1) := BattleAxe;
   Put_Line("Your items are now");
   FOR I IN 1..NumItems LOOP
      Put(MyItems(I)); New_Line;
   END LOOP;
   New_Line;
   Put_Line("You find and take a healing potion.");
   IF (NumItems < MAX_ITEMS) THEN
      BEGIN
         NumItems := NumItems + 1;
         MyItems(NumItems) := Healing_Potion;
      END;
   ELSE
      Put_Line("You have too many items and can’t carry another.");
   END IF;
   Put_Line("Your items are now");
   FOR I IN 1..NumItems LOOP
      Put(MyItems(I));
      New_Line;
   END LOOP;
   New_Line;
END;
```

4.6 Word Jumble Game

Having covered loops, strings and arrays, we now look at the game of word jumble. This is a very popular classic game. It is a puzzle game in which the computer picks a word at random from a list, then creates a jumbled up word with the letters in random order. The player must attempt to guess the word and may ask for a hint if stuck. Puzzle games are very popular and many are available on line.

We will store the words and hints in two sets of strings. We immediately think of storing the lists of words and hints in two arrays, which would be convenient because then a word and hint could be chosen at random, using the random number to set the array index, but then we realise that array elements must all be of the same type and size, whereas our strings all vary in length.

So how do we overcome this problem? One solution is to use containers, which will be explained next.

5. The Ada container library

Ada 2005 introduced the container library, a library of very powerful tools. We have seen how to handle arrays, which can have multiple dimensions and are very powerful, but have the restriction that they are fixed in size and therefor lack some flexibility. The container library provides a flexible way of handling collections of data, and this need is so common in programming that a whole section of the standard is devoted to it. The container library provides a large number of functions for handling vectors, queues, lists, maps and other data structures. These tools make it much easier to write games.

For now, we will only look at vectors, and will leave the other data structures like linked lists, etc. for a more specialised course on data structures.

Specifically, we'll look at how to:

- Use vector objects to work with sequences of values
- Use vector member functions to manipulate sequence elements
- Use iterators to move through sequences
- Use library algorithms to work with groups of elements

Introducing the Container Library

Programming is hard and it is easy to make mistakes, so good game programmers like to reuse code that they know works and has been thoroughly tested. This can make programming much easier and eliminate errors. In any case, why redo work that's already been done—especially if it has been done well? The Container Library represents a large and powerful collection of programming work that's been done well. It provides a group of containers, algorithms, and iterators, among other things.
So what's a container and how can it help you write games? Well, containers let you store and access collections of values of the same type. Yes, arrays let you do the same thing, but containers offer more flexibility and power than arrays. The container library defines a variety of container types; each works in a different way to meet different needs.

The algorithms defined in the container library work with its containers. The algorithms are common functions that game programmers find themselves repeatedly applying to groups of values. They include algorithms for sorting, searching, copying, merging, inserting, and removing container elements. The cool thing is that the same algorithm can work its magic on many different container types.

Iterators are objects that identify elements in containers and can be manipulated to move among elements. This is called iterating, and they're great for moving about, that is, iterating, through the elements of containers. In addition, iterators are required by the container library algorithms.

All of this makes a lot more sense when you see an actual implementation of one of the container types, so that's up next.

5.1 Using Vectors

The vector class defines one kind of container provided by the container library. It meets the general description of a dynamic array—an array that can grow and shrink in size as needed. In addition, the vector container defines member functions to manipulate vector elements. This means that the vector has all of the functionality of the array plus more.

At this point, you may be thinking to yourself: Why learn to use these fancy new vectors when I can already use arrays? Well, vectors have certain advantages over arrays, including:

- Vectors can grow as needed while arrays cannot. This means that if you use a vector to store objects for enemies in a game, the vector will grow to accommodate the number of enemies that get created. If you use an array, you have to create one that can store some maximum number of enemies. And if, during play, you need more room in the array than you thought, you're out of luck.
- Vectors can be used with the container algorithms while arrays cannot. This means that by using vectors you get complex functionality like searching and sorting, built-in. If you use arrays, you have to write your own code to achieve this same functionality.

There are a few disadvantages to vectors when compared to arrays, including:

- Vectors need extra memory as overhead.
- There can be a performance cost when a vector grows in size.
- Vectors may not be available on some game console systems.

Overall, vectors (and the rest of the container library) can be a welcome tool in virtually any programming project.

5.2 Introducing the Player's Inventory 2.0 Program

From the user's point of view, the Player's Inventory 2.0 program is similar to its predecessor, the Player's Inventory program from Section 5.5. Our new version stores and works with a collection of string objects that represent a player's inventory. However, from the programmer's perspective the program is quite different. That's because the new program uses a vector instead of an array to represent the inventory and will also use strings instead of an Enumeration type. Since the strings all vary in length, we have to define the vector as an indefinite vector, where the actual size of each element can be determined when the element is added to the vector.

Here is the program:

```
-- Player's Inventory 2.0 - Demonstrates vectors

WITH Text_Io; USE Text_Io;
WITH Ada.Integer_Text_IO; USE Ada.Integer_Text_IO;
WITH Ada.Containers.Indefinite_Vectors; USE Ada.Containers;

PROCEDURE PlayersInventory2 IS
   PACKAGE Word_Container IS NEW Indefinite_Vectors (Natural, String);
   USE Word_Container;
   Words : Vector;

BEGIN
```

```
   Words.Append ("sword");
   Words.Append ("armour");
   Words.Append ("shield");

   Put( "You have "); Put(Integer(Words.Length), 1); Put_Line(" items.");
   Put_Line( "Your items are:");
   FOR I IN 0 .. Integer(Words.Length)-1 LOOP
      Put(WORDS.Element (I)); New_Line;
   END LOOP;
   New_Line; New_Line;

   Put_Line( "You trade your sword for a battle axe.");
   Replace_Element (Words, 0, "battle axe");
   Put_Line( "Your items are now:");
   FOR I IN 0 .. Integer(Words.Length)-1 LOOP
      Put(WORDS.Element (Index => I)); New_Line;
   END LOOP;
   New_Line;

   Put( "The item name "); Put(Words.Element(1)); Put( " has ");
   Put( Words.Element(1)'length, 1); Put_Line( " letters in it.");
   New_Line; New_Line;

   Put( "Your shield is destroyed in a fierce battle.");
   Words.Delete_Last;
   New_Line;

   Put_Line( "Your item are now");
   FOR I IN 0 .. Integer(Words.Length)-1 LOOP
      Put(WORDS.Element (Index => I)); New_Line;
   END LOOP;

   New_Line; New_Line;
   Put( "You were robbed of all of your possessions by a thief. ");
   Words.Clear;

   IF (Words.Is_Empty) THEN  Put( "You have nothing.");
   ELSE  Put( "You have at least one item.");
   END IF;
END;
```

We see that the vector container allows us to hold strings of varying length, and we can append items, scan through the container, delete items, etc. The LRM has a complete list of the operations that can be performed on vectors.

Before we can use vectors, we have to first include the vector package, which is in the containers package: `WITH Ada.Containers.Indefinite_Vectors; USE Ada.Containers;`
As I usually do, I include `USE Ada.Containers;` so I can refer to a vector without having to precede it with ada.containers.

Okay, the first thing we do in the program body is declare a new instance of the vector container and say what kinds of things it will contain:

```
PROCEDURE PlayersInventory2 IS  -- start of program bony
   PACKAGE Word_Container IS NEW Indefinite_Vectors (Natural, String);
   USE Word_Container; -- this vector will contain strings
   Words : Vector;   -- make an instance of it
```

Now we have an empty vector named Words, which can contain string object elements of any length. Declaring an empty vector is fine because it grows in size when you add new elements. The library allows you to declare a vector with an initial size, but this is not necessary. So we had better add some strings to the vector, which we can do with the Append function:

```
   Words.Append ("sword");
```

etc.

Now we have three items in the vector. The function Words.length tells us now many there are. Of Couse, we know there are 3, so this just to show how the Words.length function works, as we often use this function in a game as items are added and removed from the inventory due to the player encountering different situations.

Now we list the items in the inventory, using words.length for the listing. Note that the first item in the inventory is at position 0, so we list from 0 to length-1:

```
   Put_Line( "Your items are:");
   FOR I IN 0 .. Integer(Words.Length)-1 LOOP
      Put(WORDS.Element (I)); New_Line;
   END LOOP;
   New_Line; New_Line;
```

Next we trade in the sword for a battle axe, using the replace_element function, and then list the items again.

```
   Put_Line( "You trade your sword for a battle axe.");
   Replace_Element (Words, 0, "battle axe");
   Put_Line( "Your items are now:");
   FOR I IN 0 .. Integer(Words.Length)-1 LOOP
      Put(WORDS.Element (Index => I)); New_Line;
   END LOOP;
   New_Line;
```

The next bit of code takes the second element, which is the string “armour” and displays its length attribute, which in this case is the number of characters in the string (as against words.length which gives the number of strings in the words container.

```
   Put( "The item name "); Put(Words.Element(1)); Put( " has ");
   Put( Words.Element(1)'length, 1); Put_Line( " letters in it.");
```

The last sections of code delete one item from the container and display the remaining items, then clear the container, which empties it of all items, and applies the Is_Empty function to confirm that it is empty.

5.3 Word Jumble

Having seen how we can set up strings using a vector container, let’s now return to designing a word jumble program using a container.

Word Jumble is a puzzle game in which the computer picks a word at random from a list, then creates a jumbled up word with the letters in random order. The player must attempt to guess the word and may ask for a hint if stuck.

We will store the words and hints as pairs of strings. We will use a single vector container, with each word followed by its corresponding hint. We set up the vector as in the Player's Inventory program above, push the words and hints onto the vector, then select a random word and corresponding hint. Since the strings all vary in length, we use an indefinite vector to store them.

We start as usual with some pseudocode:

```
list the packages we wish to use
set up the program data, namely the strings and hints and the container for accessing them
create a specific instance of the random number generator
list the variables we will use in the program
display a start up message for the player
generate random number for the choice of word and hint
select the chosen word and hint from the list of words and hints in the container. Since the
    lengths of the words and hints vary, so we don't know their lengths in advance, make
    the selection part of a declaration block so the system will create string variables
    theWord and theHint of the correct length.
jumble the selected word by swapping characters in the word at random.
Start a loop
    display the jumbled word and ask the player to guess the word.
    If the guess is correct, we congratulate the player and exit.
    If the player asks for a hint, display the hint and if he/she says quit, the program exits.
```

Here is the code, which follows the pseudocode closely

```
-- Word Jumble
-- The classic word jumble game where the player can ask for a hint

WITH Text_Io;                           USE Text_Io;
WITH Ada.Strings.Unbounded;             USE Ada.Strings.Unbounded;
WITH Ada.Numerics.Discrete_Random;
WITH Ada.Strings;                       USE Ada.Strings;
WITH Ada.Text_IO.Unbounded_IO;          USE Ada.Text_IO.Unbounded_IO;
with Ada.Containers.Indefinite_Vectors; use Ada.Containers;

procedure Jumble is
   package words_Container is new Indefinite_Vectors (Natural, String);
   use words_Container;
   Words     : Vector;
   NUM_WORDS : integer;

   PACKAGE RInt IS NEW Ada.Numerics.Discrete_Random (Integer);  USE RInt;
   G      : Generator;

   Choice : Integer;
   Guess  : Unbounded_String;
   Temp   : Character;
   Index1, Index2 : Integer;

BEGIN
   -- set up the words and hints in the string vector
   words.Append (New_Item => "wall");
   words.Append (New_Item => "Banging your head against something?");
   words.Append (New_Item => "glasses");
   words.Append (New_Item => "These might help you see the answer.");
   words.Append (New_Item => "labored");
   words.Append (New_Item => "Going slowly, is it?");
   words.Append (New_Item => "persistent");
   words.Append (New_Item => "Keep at it.");
   words.Append (New_Item => "jumble");
   words.Append (New_Item => "It's what the game is all about.");
   words.Append (New_Item => "lecture");
   words.Append (New_Item => "Lots of boring words and powerpoints.");
   num_words := integer(words.length) / 2;

   Put_Line("      Welcome to Word Jumble!");
   New_Line;
   Put_Line("Unscramble the letters to make a word.");
   Put_Line("Enter 'hint' for a hint.");
   Put_Line("Enter 'quit' to quit the game.");
   New_Line;

   Reset(G);
   Choice := Random(G) Mod NUM_WORDS;
   DECLARE
      TheWord : String := WORDS.element (index => choice * 2);
      TheHint : String := WORDS.element (index => Choice * 2 + 1);
      Jumble : String  := TheWord;
      Len     : Integer := TheWord'Length;
   BEGIN
      FOR I IN 1..Len LOOP
         BEGIN
            Index1 := (Random(G) Mod Len) + 1;
            Index2 := (Random(G) Mod Len) + 1;
            Temp := Jumble(Index1);
            Jumble(Index1) := Jumble(Index2);
            Jumble(Index2) := Temp;
         END;
      END LOOP;

      Put("The jumble is: ");
      Put(Jumble);
      New_Line;

      New_Line;
      Put("Your guess: ");
      Get_Line(Guess);

      WHILE ((Guess /= TheWord) AND (Guess /= "quit")) LOOP
         BEGIN
            IF (Guess = "hint") THEN
               Put_Line(TheHint);
            ELSE
               Put_Line("Sorry, that's not it.");
            END IF;
            Put("Your guess: ");
            Get_Line(Guess);
         END;
      END LOOP;
      IF (Guess = TheWord) THEN
         BEGIN
            New_Line;
            Put_Line("That's it!  You guessed it!");
         END;
      END IF;
```

```
      New_Line;
      Put_Line("Thanks for playing.");
   END;
```

The one tricky part in this example is the use of a declaration block to create automatically sized string variables for theWord and theHint. This is a useful feature of Ada that allows us to use standard strings instead of unbounded strings.

It would also be useful to store the words and hints on a disk file and read them in when the program starts. Later we will see how to do that.

5.4 Using Iterators

Iterators are the key to using containers to their fullest potential and Ada2012 uses iterators throughout the container library. With iterators you can, well, iterate through the items in a container in sequence. In addition, important parts of the container library require iterators. Many container member functions and algorithms take iterators as arguments. So if you want to reap the benefits of these member functions and algorithms, you've got to use iterators.

Introducing the Player's Inventory 3.0 Program

The Player's Inventory 3.0 program acts just like its two predecessors, at least at the start. The program shows off a list of items, replaces the first item, and displays the number of letters in the name of an item. But then the program does something new: It inserts an item at the beginning of the group, and then it removes an item from the middle of the group. The program accomplishes all of this by working with iterators.

In Ada, the index part of the iterator is called a cursor, a bit like the cursor on a text screen, which shows you where you are on the screen, so the container cursor shows where you are in the container. But what, then, is the difference between a cursor and an index (which we used in the two previous versions of our program). Well, the cursor is part of the iterator and it is built into the container, so it knows exactly how to work through the container without knowing anything about the internal implementation of the container, which is important for more complex structures like queues and maps.

Here is the code for version 3:

```
-- Player's Inventory 3.0 - Demonstrates iterators

WITH Text_Io; USE Text_Io;
WITH Ada.Integer_Text_IO; USE Ada.Integer_Text_IO;
WITH Ada.Containers.Indefinite_Vectors; USE Ada.Containers;

PROCEDURE PlayersInventory3 IS
   PACKAGE Word_Container IS NEW Indefinite_Vectors (Natural, String);
   USE Word_Container;
   Words : Vector;
   c : cursor;  -- type cursor is used for iteration

BEGIN
   Words.Append ("sword"); Words.Append ("armour");
   Words.Append ("shield");
   Put( "You have "); Put(Integer(Words.Length), 1); Put_Line(" items.");
   Put_Line( "Your items are:");
   FOR c IN Words.iterate LOOP   -- iterate through Words using cursor
      Put(WORDS(c)); New_Line;
   END LOOP;
   New_Line; New_Line;
   Put_Line( "You trade your sword for a battle axe.");
   c := first(Words);  -- point cursor at first item in container
   Replace_Element (Words, c, "battle axe");
   Put_Line( "Your items are now:");
   FOR c IN Words.iterate LOOP
      Put(WORDS(c)); New_Line;
   END LOOP;
   New_Line;
   Put( "The item name "); Put(Words.Element(1)); Put( " has ");
   Put( Words.Element(1)'length, 1); Put_Line( " letters in it.");
   New_Line; New_Line;
   Put( "Your shield is destroyed in a fierce battle.");
   C:= words.find(”shield”);
   Words.delete(c);
   New_Line;
   Put_Line( "Your items are now");
   FOR c IN Words.iterate LOOP
      Put(WORDS(c)); New_Line;
   END LOOP;
   New_Line; New_Line;
   Put_line( "You recover a crossbow from a slain enemy.");
   Put_Line( "You add it to your list before the armour.");
   c := Find(Words, "armour");
   insert (Words, c, "crossbow");
   New_Line; Put_Line( "Your items are now");
   FOR Elements OF Words LOOP  -- alternate form of iteration, where the
      Put(Elements); New_Line; -- iterator itself is hidden. This form
   END LOOP;                   -- returns all the elements of the container
   New_Line;
   Put( "You were robbed of all of your possessions by a thief. ");
   Words.Clear;
   IF (Words.Is_Empty)
     THEN Put( "You have nothing.");
     ELSE Put( "You have at least one item.");
   END IF;
END;
```

The program starts off exactly the same as version 2.0 but then we add a variable c or type cursor to use later. We then append items to the vector at before. When we list the items, however, we use:

```
   FOR c IN Words.iterate LOOP   -- interate through Words using cursor
      Put(WORDS(c)); New_Line;
   END LOOP;
```

We don’t need to know the length of the list, or how to move through it – the iterator takes care of that for us. We then replace the first item with:

```
   c := first(Words);  -- point cursor at first item in container
   Replace_Element (Words, c, "battle axe");
```

There are lots of other functions for moving around the vector, such as last, previous and next. They are all listed in the LRM.

Next, we delete an item (the shield) but first find it in the list:

```
   Put( "Your shield is destroyed in a fierce battle.");
   C:= words.find(”shield”);
   Words.delete(c);
```

If the item were not in the list, find would return No_Element. This could be checked using an IF statement, otherwise the delete would return an error.

Next, we insert an item into the list. Append would add it to the end, prepend to the beginning and insert will add it anywhere in the list, before the item pointed to by the cursor:

```
   c := Find(Words, "armour");
   insert (Words, c, "crossbow");
   New_Line;
```

Note that this form: `c := Find(Words, "armour");` is equivalent to the form `c := Words.Find("armour");` and the two forms can be used interchangeably.

Lastly, we show an alternate simplified form of iterating through a container, where we do not need to specify the iterator. Note the use of the keyword OF instead of IN.

```
   FOR Elements OF Words LOOP  -- alternate form of iteration, where the
      Put(Elements); New_Line; -- iterator itself is hidden. This form
   END LOOP;                   -- returns all the elements of the container
```

So now we have seen how iterators work, let’s take a look at algorithms.

5.5 Using Algorithms

The Ada containers package defines algorithms that allow you to manipulate elements in containers through iterators. Algorithms exist for common tasks such as searching, randomizing, and sorting. These algorithms are your built-in arsenal of flexible and efficient weapons. By using them, you can leave the mundane task of manipulating container elements in common ways to the container library so you can concentrate on writing your game. The powerful thing about these algorithms is that they are generic—the same algorithm can work with elements of different container types. We have already used several algorithms such as find. The major new algorithm we will use now is sort. We will show you how to use this in the next example

Introducing the High Scores Program

The High Scores program creates a vector of high scores. It uses container algorithms to search and sort the scores and displays them in low to high then in high to low order.

```
-- High Scores - Demonstrates algorithms

WITH Text_Io;                 USE Text_Io;
WITH Ada.Integer_Text_IO;     USE Ada.Integer_Text_IO;
WITH Ada.Containers.Vectors;  USE Ada.Containers;

PROCEDURE HighScores IS
   PACKAGE Scores_Container IS NEW Vectors (Natural, Integer);
   USE Scores_Container;
   Scores : Vector;
   C       : Cursor;
   PACKAGE Sorter IS NEW Generic_Sorting; -- Instance of Sort Algorithm
   USE Sorter;
   MyScore : Integer;
BEGIN
   Scores.Append (7500); --put some scores in the container
   Scores.Append (1600);
   Scores.Append (6300);
   Scores.Append (3200);
   Put( "You have created a list of scores:");
   Put_Line( "Your items are now:");
   FOR C IN Scores.Iterate LOOP   -- display all the scores
      Put(Scores(C)); New_Line;
   END LOOP;
   New_Line;
   Put( "Enter a score to find: ");
   Get(MyScore);
   C := Find(Scores, MyScore);    -- use the “find” algorithm
   IF C = No_Element THEN Put_Line("Score not found");
   ELSE  Put_Line("Score found");
   END IF;

   Put_Line("Now sorting the scores");
   Sort(Scores);   -- use the generic sorter algorithm (low to high order)
   Put_Line("The sorted scores are:");
   FOR E of Scores LOOP      -- list the score using the
      Put(E);  New_Line;     -- alternate form of iteration
   END LOOP;
   New_Line;

   Put_Line("Display in reverse order (highest first):");
   FOR E of reverse Scores LOOP   -- reverse order of iteration
      Put(E);  New_Line;
   END LOOP;
   New_Line;
END;
```

5.6 Hangman

In the Hangman game, the computer picks a secret word and the player tries to guess it, one letter at a time. The player is allowed eight incorrect guesses. If he or she fails to guess the word in time, the player is hanged and the game is over.

Before writing any code, we plan the game program using pseudocode:

```
Set up the program
Create a group of words
Welcome the player
Pick a random word from the group as the secret word
While player hasn’t made too many incorrect guesses and hasn’t guessed the secret word
Tell player how many incorrect guesses he or she has left
Show player the letters he or she has guessed and the progress in finding the secret word
Get player’s next guess
While player has entered a letter that he or she has not already guessed and has not made too many incorrect guesses
   Get player’s next guess
```

```
        Add the new guess to the group of used letters
        If the guess is in the secret word
        Tell the player the guess is correct
        Update the word guessed so far with the new letter
    Otherwise
        Tell the player the guess is incorrect
        Increment the number of incorrect guesses the player has made
        If the player has made too many incorrect guesses
        Tell the player that he or she has been hanged
    Otherwise
        Congratulate the player on guessing the secret word
```

Much of this we have already done. We start with some comments and by setting up the program by "with"ing the packages we need. Here we also include Ada.Characters.Handling because we will use the function To_Upper which is part of the package and converts characters to upper case.

```
-- The classic Hangman game

-- set up the program and the packages it uses
WITH Text_Io;  USE Text_Io;
WITH Ada.Integer_Text_IO; USE Ada.Integer_Text_IO;
WITH Ada.Numerics.Discrete_Random;
WITH Ada.Containers.Indefinite_Vectors; USE Ada.Containers;
WITH Ada.Characters.Handling; USE Ada.Characters.Handling;
```

Next we set up a vector container to hold the list of words. Once again, we use a container as it will hold variable length strings and make it easy to choose one at random. We also set up the random number package to use to select one of the words at random.

```
PROCEDURE Hangman4 IS
   PACKAGE Words_Container IS NEW Indefinite_Vectors (Natural, String);
   USE Words_Container;  -- use this for the word list
   Words     :          Vector;
   Num_Words :          Integer;
   Max_Wrong : CONSTANT Integer := 8;

   PACKAGE RInt IS NEW Ada.Numerics.Discrete_Random (Integer);
   USE RInt;
   G : Generator;
```

Now we begin the main part of the program by creating a list of words. In a real game of Hangman, the list would be a lot longer and would be stored in a disk file.

```
BEGIN
   -- Create a group of words
   Words.Append (New_Item => "GUESS");
   Words.Append (New_Item => "HANGMAN");
   Words.Append (New_Item => "GLASSES");
   Words.Append (New_Item => "PERSISTENT");
   Words.Append (New_Item => "DIFFICULT");
   Words.Append (New_Item => "LECTURE");
   Words.Append (New_Item => "BORING");
   Num_Words := Integer(Words.Length);
```

The next step is to welcome the player, with some instructions.

```
Put_Line("      Welcome to Hangman");
New_Line;
Put_Line("Try to guess the word, one letter at a time");
Put_Line("You are limited to 8 wrong guesses");
New_Line;
```

Next we create a random number, first by randomising the generator and then obtaining a random integer, followed by taking the remainder from dividing by the number of words. This gives a random number, Choice, in the range 0 .. num_words – 1. We then use this to choose one of the words from the container.

```
Reset(G); -- randomize the random number generator
DECLARE   -- the data needed by the game
   Choice : Integer := Random (G) Mod NUM_WORDS;
   --pick a random word from the group as the secret word
   TheWord : String := WORDS.Element (Index => Choice);
   -- set up some more variables needed by the program
```

The next step is to declare some more data needed by the game. So_Far will hold the letters guessed correctly so far, with the rest displayed as `'-'`. Used will keep track of the letters used. It is initialised to all blanks so the end of the list of letters used can easily be detected as a blank. IsIn is a Boolean variable, which can only take values of True or False, and will be used to indicate if a guessed character is in the word.

```
   WordLen : Integer               := TheWord'Length;
   So_Far  : String (1 .. WordLen) := (OTHERS => '-'); -- guessed so far
   Used    : String (1 .. 26)      := (OTHERS => ' '); -- letters used
   NumUsed : Integer               := 0;
   Wrong   : Integer               := 0;          -- incorrect guesses
   Guess   : Character;
   IsIn    : Boolean;
```

Now we are ready to code the game. The game will continue as long as the player hasn't made too many wrong guesses (limited to 8 by the constant Max_Wrong defined above) and hasn't guessed the word so far. When Wrong = Max_Wrong the game ends and the player is hanged (loses) and when So_Far = TheWord the player wins. These two conditions can be combined in a while loop:

```
BEGIN
   WHILE ((Wrong < Max_Wrong) AND (So_Far /= TheWord)) LOOP
      New_Line;
```

Now we continue with the body of the while loop. First, tell player how many incorrect guesses are left, display the letters used (in string Used) and the correct guesses so far (in string So_Far):

```
      Put("You have ");
      Put (MAX_WRONG - Wrong, 1);
      Put_Line(" incorrect guesses left.");
      -- Show player the letters guessed so far
      -- and the progress in finding the secret word
      Put_Line("You've used the following letters:");
      Put (Used);
      New_Line;
      Put_Line("So far, the word is:");
      Put_Line(So_Far);
```

Now get player’s next guess and make sure it is upper case (capitals) to the user can enter either lower case (small) or capital letters:

```
         New_Line;
         Put("Enter your guess: ");
         Get(Guess);
         Guess := To_Upper(Guess);
```

Next, check if the player has entered a new letter, not already guessed. The already used letters are in the string Used. If we hit the end of the string of used letters, detected by encountering the space character `' '`, we exit the loop and add the latest guess to the string Used.

```
         FOR I IN 1 .. 26  LOOP  -- 26 letters at most
            IF Used(I) = Guess THEN
               New_Line;
               Put_Line("You've already guessed " & Guess );
               Put("Enter your guess: ");
               Get(Guess);
               Guess := To_Upper(Guess);
            ELSIF Used(I) = ' ' THEN  -- end of list of used letters
               EXIT;
            END IF;
         END LOOP;

         NumUsed := NumUsed + 1;      -- number of letters used
         Used(NumUsed) := Guess;      -- save the latest guess
```

Next, we search the secret word to see if it contains the latest character guessed. If we find the guess in the word, we add it to So_Far and make IsIn true.

```
         IsIn := False;               -- now check if it is in the word
         FOR I IN 1 .. WordLen LOOP   -- check each letter in the word
            IF (TheWord(I) = Guess) THEN
               So_Far(I) := Guess;    -- yes, it is in the word, mark the
               IsIn := True;          -- letter and note we found it
            END IF;
         END LOOP;
```

We have finished checking the secret word and updating So_Far, so now tell the player if the guess was in the word or not. If it wasn’t in the word, the guess was wrong, so increment the variable Wrong, which counts wrong guesses.

```
         IF IsIn THEN
            Put("That's right! " & Guess & " is in the word.");
         ELSE
            Put_Line("Sorry, " & Guess & " isn't in the word.");
            Wrong := Wrong + 1;
         END IF;
```

This is the end of the main While loop, which will exit if the player has guessed the word or hit the limit of wrong guesses.

```
      END LOOP;
```

We’re done, so tell player if he or she guessed the word successfully or not, reveal the secret word, and then end the game.

```
      New_Line;
      IF (Wrong = MAX_WRONG) THEN
         New_Line;
         Put_Line("You've been hanged!");
      ELSE
         Put_Line( "You guessed it!");
      END IF;
      New_Line;
      Put_Line( "The word was " & TheWord);
   END;
END;
```

6. Programming your own subprograms - procedures and functions

Procedures and functions, together called subprograms, capture code that is frequently repeated. These greatly extend the power of the programming language and make for much more compact programs. It is also good practice to build up libraries of subprograms, which can then be reused. In Ada, these are bundled together into packages, which we will talk about later, but we have already seen how Ada itself makes heavy use of all kinds of packages. For now we note that Ada defines both procedures and functions, together called subprograms.

Every program you've seen so far has consisted of one subprogram, a procedure that specifies the name of the program. Once your programs reach a certain size or level of complexity, it becomes hard to work with them like this. Fortunately, there are ways to break up big programs into smaller, bite-sized chunks of code that are easier to manage. In this chapter, you'll learn about one way—creating new subprograms. Specifically, you'll learn to:

- Write new functions
- Accept values into your new functions through parameters
- Return information from your new functions through return values

We will also touch on working with global variables and constants, overloading subprograms and inlining subprograms.

For both, information is passed to the procedure or function by means of arguments, and for a function, a result to be returned is also defined. In Ada, functions must return a result, procedures don't have to.

We have already used many procedures and functions:

```
      Put(anInteger, width => 5);  a procedure to display an integer
```

This procedure also has two arguments, the value to be displayed, and the number of columns to be used to display the value. Here is another procedure and functions we have used:

```
      Get(Number);   -- procedure to read a number from the keyboard
      Res := index(String1, String2); -- function to return the index in
                -- String 1 if String 2 is present in String1, else 0.
```

Ada provides large number subprograms in its library of packages, for a variety of both general and specialised requirements, as we noted at the beginning of this tutorial. The Text_IO package is used for screen and keyboard work, the Strings package for manipulating strings, etc. Full details of all the packages are given in the LRM. A package that is very widely used is Numerics, which offers subprograms like sin, cos, sqrt, etc. There are also packages for complex arithmetic and for real and complex vector and matrix manipulation. We have used lots of functions and procedures from several packages already, such as the random function from the discrete random package, append, find and sort from the vector containers package, etc. Now it's time to learn to write your own.

6.2 Writing your own subprograms

Ada lets you write programs with multiple subprograms. Your new subprograms work just like the ones that are part of the standard language—they go off and perform a task and then return control to your program. A big advantage of writing new subprograms is it allows you to break up your code into manageable pieces. Just like the subprograms you've already learned about from the standard library, your new subprograms should do one job well.

We'll start by defining some procedures.

Before you can call a procedure you've written, you have to specify it. The specification uses the keyword Procedure, followed by its name, then a list of parameters between a set of parentheses. Parameters receive the values sent as arguments in a procedure call. So, to write you own procedure, you first say:

```
Procedure name(list of parameters);
```

This part is called the specification. Subprograms are defined in two parts, a specification and a body. The *specification* defines how the subprogram is called and the *body* defines what the subprogram does when it is called.

Specifications are not the only way to declare a procedure. Another way to accomplish the same thing is to let the procedure definition act as its own specification. To do that, you simply have to put your procedure code before the call to the procedure.

To write you own procedure, you first say:

```
Procedure name(list of parameters);
```

The specification is required when the body is in a separate library and provides the definition of the interface to the subprogram. It is not required if the subprogram is written as part of a program in a single file. So, although you don't always have to use separate specifications, they offer a lot of benefits—not the least of which is making your code clearer.

The *list of parameters* defines the name, type and mode of each of the parameters. The modes are in, out, in out and specify if the argument values are passed to (in), from (out) or in both directions (in out) between caller and callee. For example, here is a specification of a procedure with one parameter:

```
Procedure displayVal(val: integer);
```

The name and type of the parameters must always be given. The mode defaults to In if not specified. As an example, the Get procedure in Text_IO uses an Out parameter to return the item read from the keyboard.

The body of the procedure then follows:

```
Procedure displayVal(val: integer) is
BEGIN
      <code to print the specified Val>
END;
```

Calling the procedure is simple. You call your own procedures the same way you call any other procedure—by writing the procedure's name followed by a pair of parentheses that encloses a valid list of arguments. You call your newly minted procedure simply with:

```
displayVal(SomeVal);
```

This line calls displayVal and passes to it the value of the data in the parameter SomeVal. Whenever you call a procedure, control of the program jumps to that procedure. In this case, it means control jumps to displayVal and the program executes the procedure's code, which displays the data and any other output in the procedure code. When the procedure finishes, control returns to the calling code.

To see how all this works, let's now create a program with two procedures, Hello, which greets a player, and Goodbye, which displays a farewell message. The results of this program are pretty basic—a few lines of text that greet the player, and a few lines of output to say goodbye. This looks like a program you could easily have written back in Chapter 1, but this program has a fresh element working behind the scenes—two new procedures. The main program, which we will call HelloGoodbye will ask for the name of the player, then call the two procedures and display the message.

We define the procedure Hello as:

```
PROCEDURE Hello (Name : String) IS
BEGIN
-- rest of the code
END;
```

It takes one parameters, a string called Name. The parameters can be any legal Ada types. Goodbye, however, has no parameters:

```
PROCEDURE Goodbye IS
```

A procedure can have no parameters, one, two or many parameters, as many as needed, and they can all be different types. Here is the code:

```
with text_io; use text_io;

procedure HelloGoodbye is

yourName : string(1..50);     -- declares a string variable
Length : Integer;       -- declares an integer variable

   PROCEDURE Hello (Name : String) IS
   BEGIN
      new_line;
      Put("Welcome to Ada, " ); Put(Name); New_Line;
      Put("We hope you are enjoying Ada"); New_Line;
   END;

   PROCEDURE Goodbye IS
   BEGIN
      new_line(2);
      Put_line("Thanks for playing." );
      Put_line("Bye for now.");
   END;

Begin  -- this is the start of the main program
   Put("Please type in your name: " );
   Get_Line(YourName, Length);
   Hello(YourName(1..length));
   GoodBye;
END HelloGoodbye;
```

Games should always start with a welcome and some instructions for the players, and end with a farewell message and a signoff. It is convenient for these to be captured in separate procedures.

6.3 Functions

Functions are the second kind of subprogram in Ada. Some other languages which only have functions, but Ada has both procedures and functions. A function, as mentioned above, always must return a value, which is then saved in a variable or used, e.g. as a parameter to another subprogram or in an IF statement. The specification gives the name, the parameters and the return type.

Let's add a function to the HelloGoodbye program, which asks if the player a question and returns a Boolean value TRUE if the player says 'y'. The specification looks like this:

```
FUNCTION AskYesNo (Question : String) RETURN Boolean;
```

The calling code might save the result, or use it in an IF statement, for example:

```
IF AskYesNo (“Do you want to play”) THEN  -- Etc.
```

Here is the code for an extended program, HelloGoodByeV2, which includes AskYesNo:

```
with text_io; use text_io;

procedure HelloGoodbyeV2 is

yourName : string(1..50);     -- declares a string variable
Length : Integer;       -- declares an integer variable

   PROCEDURE Hello (Name : String) IS
   BEGIN
      new_line;
      Put("Welcome to Ada, " ); Put(Name); New_Line;
      Put("We hope you are enjoying Ada"); New_Line;
   END;

   PROCEDURE Goodbye IS
   BEGIN
      new_line(2);
      Put_line("Thanks for visit ing." );
      Put_line("Bye for now.");
   END;

   FUNCTION AskYesNo (Question : String) RETURN Boolean IS
            Reply : Character;
   BEGIN
      LOOP
         Put(Question); Put( " (y/n): ");
         Get_Immediate(Reply); Put(Reply);
         New_Line;
         EXIT WHEN Reply IN 'Y' | 'y' | 'N' | 'n';
      END LOOP;
      RETURN Reply IN 'Y' | 'y'; -- TRUE if Y or y, else False
    END;

Begin  -- this is the start of the main program
   Put("Please type in your name: " );
   Get_Line(YourName, Length);
   Hello(YourName(1..Length));
```

```
      IF AskYesNo ("Do you want to play") THEN
         Put_Line("Sorry, the game is not ready yet.");
         end if;
      GoodBye;
   END HelloGoodbyeV2;
```

For the function AskYesNo, I output the string provided, followed by asking for a y/n answer. I have included a test for a valid input and made the function loop until the player types in a valid response. This is always a wise thing to do with user input and users often make typing errors. Notice the use of the use of IN in the IF statement, which allows a program to check if a variable or expression belongs to a range or list of items (Ada calls this a membership choice). In the main program I then call AskYesNo as part of an IF statement to check the user's reply.

6.4 More functions

Let's now look an another simple function, cubeRoot.

```
   Function cubeRoot(val: float) return Float;
```

Cube roots are sometimes needed in graphics operations, but a cube root function is not included in the standard numerics library as any root can be calculated using logs and exponentials.

The body of the function then follows:

```
   Function body cubeRoot(val: float) return float is
   BEGIN
         Return exp(log(val)/3.0);
   END;
```

This function needs some more work as it will not handle an argument <= 0. Here is a more detailed version, since the log of a negative number is undefined:

```
FUNCTION CubeRoot(Val: Float) RETURN Float IS
BEGIN
   IF Val < 0.0 THEN RETURN -Exp(Log(-Val)/3.0);
   elsif val = 0.0 then return 0.0;
   ELSE RETURN Exp(Log(Val)/3.0);
   END if;
END;
```

In all cases, the execution of a return statement terminates the function. The returned value may then be saved into a variable, so a call to the function would like:

```
Res := CubeRoot(FNum);  -- Res is a variable, FNum can be a constant.
```

Program **CubeRoots.adb** in the examples uses this function to print a table of cube roots, along with an accuracy check. Note that floats have a precision of about 6 digits, so the results for larger numbers show small errors in the 7th significant digit. For greater accuracy, instead of floats, you could use long_floats, which have a precision of 11 digits. You must then also use the matching packages Ada.Long_Float_Text_IO and Ada.Numerics.Long_Elementary_Functions. An example is given in program LongCubeRoot.adb in the examples.

Now let’s define a program that draws a varying number of faces of slightly different types one above the other to form a totem pole. This example is due to Mike Kamrad, from his great tutorial on Ada 95, with thanks. The program will do the following:

- Ask how many faces you want on the totem pole
- For the number of faces asked for loop
    - Call a function to make a random choice of face from 3 possible types of face
    - Call a procedure to display each face. It will draw faces similar to example DrawFace

To draw the face, we can modify program DrawFace.ada from section 1.4 and turn it into a procedure. First we will make an enumeration type, Faces (Happy, Surprised, Mad) so we can call the procedure with a variable of that type:

```
 type Faces is (Happy, Surprised, Mad);

  procedure Print_Face (New_Face: Faces) is
  begin
     Ada.Text_Io.Put_Line ("\\\\\\\\\\");
     Ada.Text_Io.Put_Line ("|(*) | (*)|");
     Ada.Text_Io.Put_Line ("|    O    |");
     case New_Face is
        when Happy =>
           Ada.Text_Io.Put_Line ("\_________/");
        when Surprised =>
           Ada.Text_Io.Put_Line ("\    0    /");
        when Mad =>
           Ada.Text_Io.Put_Line ("\Xxxxxxxxx/");
     end case;
  end Print_Face;
```

Now we want a function to generate one of the 3 types of face at random. We can create a random number in the range 0..2 and convert it to a face with face’val(number). We can of course use the discrete random package to do this, but for this example, just to show a different approach, we will create the random number from the system clock, as the time will differ each time the program is run. For this, we will introduce another package, ada.real_time, which provides a high resolution time of day clock along with several functions to handle time points and time intervals. We can read the time and convert it to seconds and nanoseconds using the real-time functions in the LRM, section D-8. We will take the nanosecond time span, convert it to 10 nS steps and take the remainder when divided by 3, convert that to type faces and return it. This function requires no parameters.

```
  With ada.real_time; use ada.real_time;
  FUNCTION Make_A_Face RETURN Faces IS
     SC : seconds_count; TS : time_span;
     TenNS : integer;
     BEGIN
       Split(Clock, SC,  TS);  -- read the clock, convert to time_span
                               -- to make a random number
       TenNS := TS/Nanoseconds(10); -- number of 10 nS steps in TS
       return Faces'Val(TenNS mod 3); -- return Happy, Surprised or Mad
  end Make_A_Face;
```

Now we have a procedure to draw 3 kinds of faces and a function to select one of three faces at random, we can write the main program. The complete program is in the examples.

```
WITH Ada.Integer_Text_Io; USE Ada.Integer_Text_Io;
with Ada.Text_Io; use Ada.text_Io;

PROCEDURE TotemPole IS
   TYPE Faces IS (Happy, Surprised, Mad);
   Totems : Integer;

   PROCEDURE Print_Face(New_Face : Faces); -- subprogram specs
   FUNCTION Make_A_Face RETURN Faces;

BEGIN
   Put("How Many Faces in The Totem Pole? ");
   Get (Totems);
   New_Line;
   Put("Here is A Totem Pole with " );
   Put(Totems, width => 2);
   Put_Line (" Faces");
   New_Line (2);
   FOR Current_Face IN 1..Totems LOOP
      Print_Face(Make_A_Face);
   END LOOP;
END;
```

6.5 Function to calculate and return the GCD of two numbers

Let’s now look at how an earlier example, GCD, could be made into a function. The GCD algorithm calculates the Greater Common Divisor of two positive integers and returns a positive integer:

```
FUNCTION GCD(R0, R1 : integer) RETURN Integer IS
BEGIN
   LOOP
      IF R0 = R1 THEN RETURN R0;
      ELSIF R0 > R1 THEN R0 := R0 - R1;
      ELSE R1 := R1 - R0;
      END IF;
   END LOOP;
END;
```

This function should raise an exception if R0 or R1 were <= 1. Try this as an exercise. (Hint – use the predefined type positive.

6.6 Dice game of Craps

As another example of calling a function, here is a simplified version of the game of Craps

Craps is a popular casino dice game where the player rolls two dice. This version will be simplified for one player, with no betting. Each round has two phases: Come Out and Point. To start a round, the shooter makes a Come Out roll. A Come Out roll of 2, 3 or 12 (called Craps, the shooter is said to 'crap out') ends the round with the player losing. A Come Out roll of 7 or 11 (a Natural) is a win. A come out roll of 4, 5, 6, 8, 9 or 10 becomes the Point. If the shooter rolls the point number, the result is a win. If the shooter rolls a seven (a Seven-out), the shooter loses and the round ends.

Since we will be rolling two dice, we define a subtype spots with range 1..6 and a second subtype points with a range 2..12 to represent the points thrown. Then we will make an instance of the random number generator to return spots (i.e. a number in the range 1..6). Next we will build a function, Roll_two, which simulates the dice roll by generating two random numbers, displaying their values and returning their sum as points.

The main program will call this function to get the results of a throw, the will follow the rules above. I will first write this as peudocode:

```
Throw the come out roll of the dice
If the throw is 2, 3 or 12, you lose
Otherwise  if the roll is 7 or 11, you win
Otherwise the roll is the point
Loop
    Roll the dice
    If the roll = the point, you win
    If the roll = 7, you lose
End loop
End of the game
```

Here is the code:

```
-- craps.adb:  Play one game of craps, gambling game using two dice

with Ada.Text_IO, Ada.Integer_Text_IO, Ada.Numerics.Discrete_Random;
USE Ada;

procedure Craps is

    subtype Spots is Integer Range 1 .. 6;
    subtype Points is Integer Range 2 .. 12;

    package Random_Package is new Ada.Numerics.Discrete_Random (Spots);
    Seed: Random_Package.Generator;

    function Roll_Two return Points is
       First, Second: Spots;
    begin
       First  := Random_Package.Random (Seed);
       Second := Random_Package.Random (Seed);
       Text_IO.Put ("Rolling ... ");
       Integer_Text_IO.Put (First, Width=>3);
       Integer_Text_IO.Put (Second, Width=>3);
       Text_IO.New_Line;
       return (First + Second);
    end Roll_Two;

    Initial, Total: Points;

 Begin  -- the main procedure
    Random_Package.Reset (Seed); -- set seed different every time
    Initial := Roll_Two;  -- "come out" roll
    case Initial is
       when 7|11   => Text_IO.Put_Line ("Natural win!");
       when 2|3|12 => Text_IO.Put_Line ("Lose!");
       when others =>
         Text_IO.Put_Line ("Point established");
         loop
            Total := Roll_Two;
            exit when Total = Initial or else Total = 7;
         end loop;
         if (Total = 7) then
            Text_IO.Put_Line ("Player craps out.  Loses!");
         else
            Text_IO.Put_Line ("Player makes the point.  Wins!");
         end if;
    end case;
 end Craps;
```

Exercise: extend the game by adding betting.

6.7 Pick the Cup, or Thimblerig

Here is another fun example, the game of Pick the Cup or ThimbleRig, a common street game, usually played by hustlers trying to relieve punters of their cash! The goal of the game is to try to guess which cup the pea is under.

Here is the code:

```
WITH Ada.Text_IO; USE Ada.Text_IO;
WITH Ada.Numerics.Discrete_Random;
WITH Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

PROCEDURE WherePea IS

SUBTYPE Cups IS Integer RANGE 1..3;
Games, Won : Integer := 0;
Pea : Cups;
Choice : Natural;

PROCEDURE Instructions IS
BEGIN
   Put_line( "This game is called Pick the Cup, or Thimblerig. The goal of
the game" );
   Put_line( "is to try to guess under which cup the pea is hidden." );
   new_line(2);
end;

FUNCTION Whereis RETURN natural IS
-- Ask user where is the pea (L, R, C or Q)
   c : character;
BEGIN
   LOOP
     put_line( "Pick the cup with the pea, any cup! [] [] []" );
     put( "L for left, C for center, and R for right (Q to quit) and press
enter: ");
     Get(C);
     new_line(2);
     CASE C IS
        WHEN 'l' | 'L' => RETURN 1;
        WHEN 'c' | 'C' => RETURN 2;
        WHEN 'r' | 'R' => RETURN 3;
        WHEN 'q' | 'Q' => RETURN 0;
        WHEN OTHERS =>
           new_line(3);
           put_line( "Sorry, that was not a valid input." );
     END CASE;
   END LOOP;
```

```
END;

PROCEDURE Showpea(Pea : cups) IS
BEGIN
   New_Line;
   put( "Here is the pea ");
   case pea is
      when 1 => put_line( "[*] [] []" );
      when 2 => put_line( "[] [*] []" );
      when 3 => put_line( "[] [] [*]" );
   END CASE;
END;

package CupNum is new Ada.Numerics.Discrete_Random(Cups);
USE CupNum;
   G : Generator;
   C : Character;
BEGIN
   reset (G);  -- initiaise random number generator
   instructions;  -- Give user some instructions
   loop
      choice := whereis;  -- Ask user where is the pea
      exit when choice = 0;  -- exit game loop if user wants to quit
      games := games + 1;
      pea := random(G);       -- pea = 1, 2 or 3
      if pea = choice then
        put_line( "Congratulations, you won!" );
        New_Line;
        won := won + 1;
      else
        Put_Line ( "Sorry, you lost.");
        New_Line;
      end if;
      showpea(pea);           -- Show pea to user
      New_Line(2);
      Put("Pause "); Get_immediate(C);    -- Wait for user to hit a key
      new_line(3);
      Put( "You won " ); put( won, 1 ); put (" of ");
      put( games, 1 ); put_line( " games" );
end;
```

This program uses 3 subprograms

- Instructions (for the user)
- ShowPea to display which cup the pea was under
- Whereis to ask the user which cup he/she thinks the pea is under
- It also uses the discrete random package to choose a cup.

The main program gives the instructions, asks the user where the pea is and tells the user if he/she won. It also keeps track of the games played and the number of time the user won and reports the result at the end. A key to writing good games is to manage the complexity of the game by breaking it down into component parts and putting the parts into subprograms

6.8 Engineering software with subprograms

One of the most important features of any programming language is the ability to break down, or factor, a program into subprograms. Let’s now talk a little about designing with subprograms.

We have already seen, in the totem pole example, how we might factor a program into subprograms. This process is one of abstraction.

6.9 Abstraction

Abstraction is the process of thinking of a software design in an holistic way, of reducing it to a set of essential characteristics in order to manage its complexity. It is one of several essential aspects to design, alone with encapsulation and inheritance, which we will discuss in more detail later.

Abstraction is very widely used in the real world. For example, if we are talking about cars, we can think of sports cars, saloon cars, convertibles, etc., and will know immediately from the abstract category what we are talking about, without specifying full details. Ada supports this approach through its use of libraries of software, captured in packages.

In game design, abstraction is used a great deal. For example, a character may be broadly described with statements like “when the player kills an enemy, the player gets some new powers based on the things the enemy could do”. As the game design is filled out, more detail may be added, such as maths for movement and game play elements in the form of graphs or tables. This process is often called stepwise refinement, with detail being added in steps until the code level is reached

6.10 Encapsulation

Encapsulation refers to the way in which Ada presents the specification (or interface) of subprograms, separate from the subprograms bodies (code, or implementation). The user only needs the specification in order to use the subprogram; the details of the body can remain hidden. In the Ada Language Reference Manual (LRM), only specifications are given; the Ada system vendors are free to implement the subprograms as they wish. Most subprograms in the Ada system are part of a package. For example, the Put subprogram is specified in the text_io package, which is described as follows: the package Text_IO provides facilities for input and output in human-readable form. For example, a part of the spec is **`procedure Put(Item : in  String);`** which accepts a string and writes it to the display.

Let’s now take a look at how we can write our own subprograms. Subprogram bodies are written using a largely classical approach, as we have seen in the examples presented above.

However, Ada allows, but does not require, the subprogram declaration to be separated from the subprogram body. For example, the subprogram specification for the cube root function:

```
Function cubeRoot(val: float) return Float;
```

may appear grouped with other subprograms, variables, constants, and type declarations in a given declarative part, while its body may appear later, in the list of bodies for the declarative part.

The main reason for such separation is readability. If the body and the specification appear together, the potentially large body is mixed with the much smaller interface specification. The specification provides the information needed to call the subprogram; it may be hard for the reader to find, especially when examining a program with a large number of subprograms spread over several pages of text.

These inconveniences are avoided in Ada. The programmer is provided with the ability to group subprogram declarations within a small space of text, so as to provide an immediate overview of all subprograms that are local to (part of) a given program unit. The separation of the subprogram declaration from its body is merely a convenience for large subprograms; but it is a necessity for subprograms declared in the visible part of a package, and for subprograms that are mutually recursive. However, requiring a split in all cases, including small subprograms, would just add verbosity without compensating advantages. In Ada, the decision to separate specs and bodies is therefore left to the programmer, except in the cases just mentioned where it is necessary.

Although this decision is left to the programmer, no semantic problems are involved since the information provided by the subprogram declaration is repeated in full in the subprogram body, and the two specifications must agree.

A second major advantage to separating subprogram specifications from their bodies is that this allows program designers to provide subprogram specifications and then leave the implementation of the bodies to the programming team. It also allows different bodies to be used without changing the rest of the code. As an example, the goto_XY(x,y) procedure positions the cursor on the display screen to the position given by the arguments x and y. The program may be run on windows, or on linux. The spec is the same on both, but will use different bodies, since the screen interfaces differ, so the Towers of Hanoi example, which uses the goto_XY procedure, can run on either windows or linux, provided the correct body is supplied for the procedure.

This ability to separate the specification and body supports **information hiding**, where the user need only be supplied with the specification (interface) and need not bother about the details of the implementation. We have seen this in practice already in using subprograms from the various library packages without caring at all about their implementation.

For now, let’s talk a bit about techniques for building good software.

6.11 Understanding Software Reuse

You can reuse subprograms in other programs. For example, since asking the user a yes or no question is such a common thing to do in a game, you could create an askYesNo function and use it in all of your future game programs. So writing good functions not only saves you time and energy in your current game project, but it can save you effort in future ones, too.

Real World: It’s always a waste of time to reinvent the wheel, so software reuse—employing existing software and other elements in new projects—is a technique that game companies take seriously to heart. The benefits of software reuse include:

1. Increased company productivity. By reusing code and other elements that already exist, such as a graphics engine, game companies can get their projects done with less effort.
2. Improved software quality. If a games company already has a tested piece of code, such as a networking module, then the company can reuse the code with the knowledge that it’s been thoroughly tested, and hopefully free of bugs.
3. Improved software performance. Once a game company has a high-performance piece of code, using it again not only saves the company the trouble of reinventing the wheel, it saves them from reinventing a less efficient one.

You can reuse code you've written by copying from one program and pasting it into another, but there is a better way. You can divide up a big game project into multiple files. You'll learn about this technique in Chapter 10, "Inheritance and Polymorphism: Blackjack."

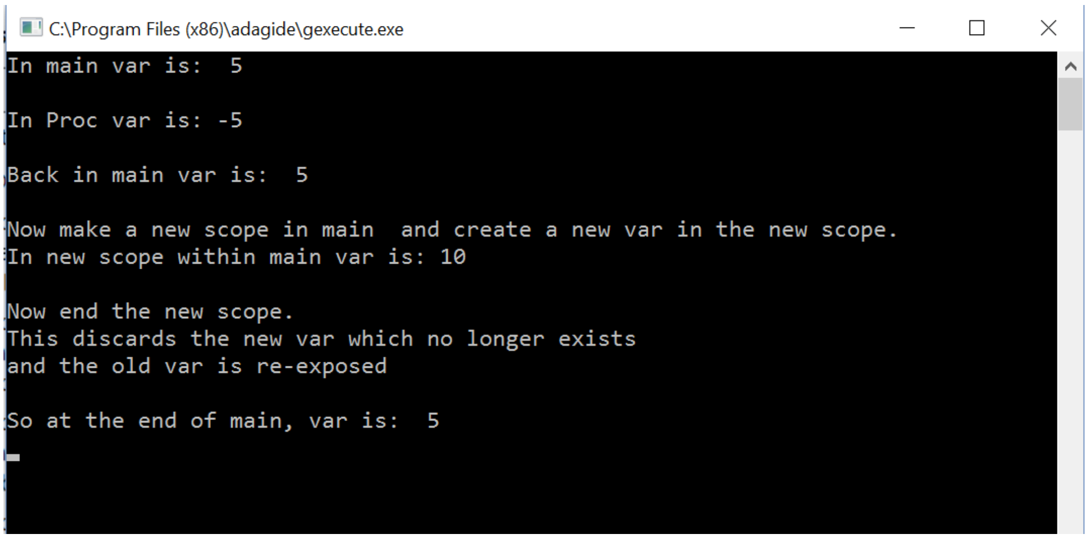
6.12 Working with Scopes

A variable's scope determines where the variable can be seen in your program.

Scopes allow you to limit the accessibility of variables and are the key to encapsulation, helping keep separate parts of your program, such as functions, apart from each other.

Introducing the Scoping Program

The Scoping program demonstrates scopes. The program creates three variables with the same name in three separate scopes. The program displays the values of these variables, and you can see that even though they all have the same name, the variables are completely separate entities. Figure 5.3 shows the results of the program.



```
In main var is:  5

In Proc var is: -5

Back in main var is:  5

Now make a new scope in main  and create a new var in the new scope.
In new scope within main var is: 10

Now end the new scope.
This discards the new var which no longer exists
and the old var is re-exposed

So at the end of main, var is:  5
```

The code for this program is in TestScopes.adb and reproduced below:

```
-- Scoping -- Demonstrates scopes

WITH Text_Io;                USE Text_Io;
WITH Ada.Integer_Text_IO; USE Ada.Integer_Text_IO;

PROCEDURE TestScopes IS

   PROCEDURE Proc IS
      Var : Integer := - 5; -- local variable in Proc
   BEGIN
      Put("In Proc var is: ");
      Put(Var, 2); New_Line; New_Line;
   END;

   Var : Integer := 5; -- local variable in TestScopes
BEGIN
   Put("In TestScopes Var is: ");
   Put(Var, 2); New_Line; New_Line;
   Proc; -- now call procedure Proc

   Put("Back in TestScopes Var is: ");
   Put(Var, 2); New_Line; New_Line;

   Put_Line("Make new scope in TestScopes, create new Var in the new
scope.");
   DECLARE
      Var : Integer := 10; -- New var in new scope, hides previous var
   BEGIN
      Put("In new scope within TestScopes Var is: ");
      Put(Var, 2); New_Line; New_Line;

      Put_Line("Now end the new scope.");
      Put_Line("This discards the new Var which no longer exists");
      Put_Line("and the old Var is re-exposed");
   END;

   new_line;
   Put("So at the end of TestScopes, Var is: ");
   Put(Var, 2); New_Line;
END;
```

6.13 Working with Separate Scopes

Every time you use BEGIN to create a block, you create a scope. Functions and procedures are examples of this. Variables declared in a scope aren't visible outside of that scope. This means that variables declared in a subprogram aren't visible outside of that subprogram. On the other hand, variables declared in an outer scope may be visible in the inner scope, as will be seen later.

Variables declared inside a subprogram are considered local variables—they're local to that subprogram. This is what makes functions encapsulated.

You've seen many local variables in action already. I define yet another local variable in the main procedure TestScopes with:

```
 var : integer = 5;  -- local variable in TestScopes
```

This line declares and initializes a local variable named var. Its fully qualified name is scopes.var, because it is part of the procedure scopes, but we can call it just var and send the variable to the display using put in the next lines of code:

```
  Put("In Proc var is: ");
  Put(Var, 2); New_Line; New_Line;
```

This works just as you'd expect: 5 is displayed.

Next I call the procedure Proc. Once I enter the procedure, I'm in a separate scope outside of the scope defined by the main procedure. As a result, I can't access the variable var that I had defined in the main procedure. This means that when I next define a variable named var in Proc with the following line, this new variable is completely separate from the variable named var in Scopes.

```
    Var : Integer := -5; -- local variable in Proc
```

Its fully qualified name is Proc.Var since it is part of the procedure Proc. Scopes.Var and Scopes.Var are different, just like, with people's names, Fred James and Fred Henry are different people, even if both are often just called Fred. If you want to be sure of addressing the right person, you would use their full name. Similarly, the two variables, Scopes.Var and Proc.Var have no effect on each other, and that's the beauty of scopes, even when, as here, we call them both just Var. So, when you write a function or a procedure, you don't have to worry if another subprogram uses the same variable names.

Then, when I display the value of Var in Proc with the following line, the computer displays -5.

```
    Put("In Proc var is: ");
    Put(Var, 2); New_Line; New_Line;
```

That's because, as far as the computer can see in this scope, there's only one variable named Var—the local variable I declared in this procedure.

Once a scope ends, all of the variables declared in that scope cease to exist. They're said to go out of scope. So next, when Proc ends, its scope ends. This means all of the variables declared in Proc are destroyed. As a result, the Var I declared in Proc with a value of -5 is destroyed. After Proc ends, control returns to the main program and picks up right where it left off.

Next, the following is executed, which sends var to the display.

```
  Put("Back in Scopes Var is: ");
  Put(Var, 2); New_Line; New_Line;
```

and the value of the var local to the main procedure Scopes (ie 5) is displayed again.

You might be wondering what happened to the Var I created in Scopes while I was in Proc. Well, the variable wasn't destroyed because Scopes hadn't yet ended. (Program control simply took a small detour to Proc.) When a program momentarily exits one subprogram to enter another, the computer saves its place in the first subprogram, keeping safe the values of all of its local variables, which are reinstated when control returns to the first function. In Ada, the main program is itself a procedure, in the example called scopes, and is itself a subprogram.

Proc has no parameters, but in subprograms that do have parameters, the parameters act just like local variables in subprograms.

6.14 Working with Nested Scopes

You can create a nested scope with Declare and a Begin/End block within in an existing scope. That's what I do next, with:

```
  Put_Line("Make new scope in Scopes, create new Var in the new scope.");
  DECLARE
    Var : Integer := 10; -- New var in new scope, hides previous var
  BEGIN
    Put("In new scope within Scopes var is: ");
    Put(Var, 2); New_Line; New_Line;
```

This new scope is a nested scope in the main program. The first thing I do is display a message that I am creating a new nested scope. According to the rules of Ada, the new scope starts with Declare, and then I declare the new variable var and initialise it to 10. If a variable hasn't been declared in a scope, when it is used the computer looks up the levels of nested scopes one at a time to find the variable you requested.

Now, with the newly declared var, when I output var to the display, 10 appears. The computer doesn't have to look up any levels of nested scopes to find var; there's a var local to this scope. And don't worry, the var I first declared in Scopes still exists; it's simply hidden in this nested scope by the new var. It can be accessed if need be using it's fully qualified name, scopes.var, which identifies the version of var declared in the main procedure scopes. This works like this

```
  Put("The previous var in the main procedure Scopes is: ");
  Put(scopes.var, 2); New_Line; New_Line;
```

T r a p -- Although you can declare variables with the same name in a series of nested scopes, it's not a good idea because it can lead to confusion. If you are not sure which is which, use the fully qualified name, Scopes.Var

Next, I end the block with:

```
    Put_Line("Now end the new scope.");
    Put_Line("This discards the new var which no longer exists");
    Put_Line("and the old var is re-exposed");
  END;
```

When the nested scope ends, the var that was equal to 10 goes out of scope and ceases to exist. However, the first var I created is still around, so when I display var for the last time in Scopes, the program displays 5.

```
  new_line;
  Put("So at the end of Scopes, var is: ");
  Put(Var, 2); New_Line;
END;
```

So, in summary, every time a new block is started, new local variables may be created. If one of these has the same name as another variable already declared, the new local variable hides the existing variable.

6.15 Parameter modes

Through the magic of encapsulation, the subprograms you've seen are all totally sealed off and independent from each other. The only way to get information into them is through their parameters, and the only way to get information out of them is to use a function with a return value. Well, that's not completely true. Subprogram parameters may be given modes of Out and In Out

The normal default mode for a parameter is In. The three modes work like this:

- `in' - the parameter's value is passed to the subprogram where it may be used but not changed.
- `out' - the parameter's value may be set in the subprogram and is passed back to the calling program.
- `in out' - the parameter's value may be used and/or changed and the updated parameter is passed back to the calling program.

Here is a an example of using an in out parameter in a simple procedure:

```
WITH Ada.Integer_Text_IO; USE Ada.Integer_Text_IO;
WITH Text_IO; USE Text_IO;

PROCEDURE Testinc IS
   Num : integer;

   PROCEDURE Increment( X : IN OUT Integer) IS
      BEGIN
         X := X+1;
      END;

BEGIN
   Put("Enter a number: ");
   Get(Num);
   put("Incremented: ");
   Increment(Num);
   Put(Num, 2);
END;
```

It is good programming practice to use a function if returning only one value and, when using functions, to use only In parameters, unless absolutely necessary.

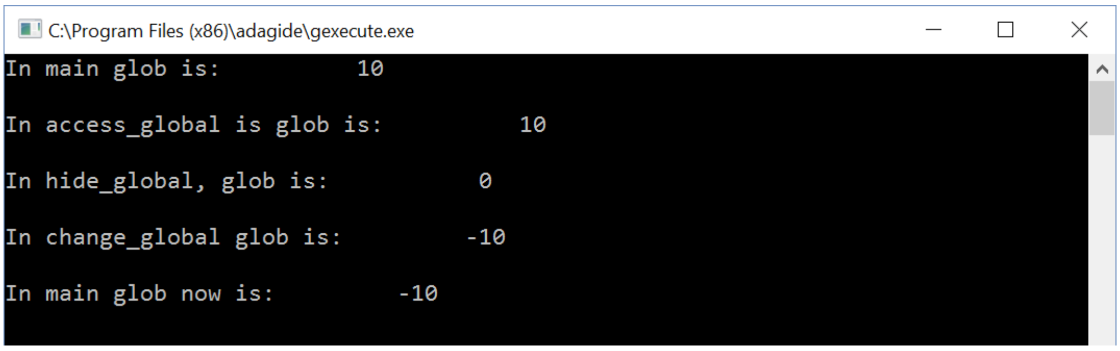
Using out or in out parameters are not the only ways to get data out of subprograms. You can also use global variables.

6.16 Using Global Variables

There is another way to share information among parts of your program—through global variables (variables that are accessible from any part of your program).

Introducing the Global Reach Program

The Global Reach program demonstrates global variables. The program shows how you can access a global variable from anywhere in your program. It also shows how you can hide a global variable in a scope. Finally, it shows that you can change a global variable from anywhere in your program. Here are the results of the program.



```
In main glob is:          10

In access_global is glob is:          10

In hide_global, glob is:          0

In change_global glob is:         -10

In main glob now is:         -10
```

The program is in the Chapter 5 folder; the filename is global_reach.cpp.

The program begins with the usual text and integer IO packages, declares the start of the program and then declares the global variable Glob and initialises it to 10.

Because glob is declared in the main procedure Global_Reach, and not in a subprogram, it is accessible both throughout the main procedure as well as in all the subprograms.

You can access a global variable from anywhere in your program. To prove it, I display glob in the main procedure with:

```
  Put( "In main glob is: ");  Put(Glob); New_Line; New_Line;
```

The program displays 10 because as a global variable, glob is available to any part of the program. To show this again, I next call access_global, and the computer executes the following code in that function:

```
    Put( "In access_global glob is: ");  Put(Glob); New_Line; New_Line;
```

Again, 10 is displayed. That makes sense because I'm displaying the exact same variable in each function.

You can hide a global variable like any other variable in a scope; you simply declare a new variable with the same name. That's exactly what I do next, when I call hide_global. The key line in that function doesn't change the global variable glob; instead, it creates a new variable named glob, local to hide_global, that hides the global variable.

```
  glob : integer := 0;  -- hide global variable glob
```

As a result, when I display glob next in hide_global with the following line, 0 is displayed.

```
    Put( "In hide_global glob is: ");  Put(Glob); New_Line; New_Line;
```

The global variable glob remains hidden in the scope of hide_global until the function ends. To prove that the global variable was only hidden and not changed, next I display glob back in main with:

```
  Put( "Back in main glob still is: ");   Put(Glob); New_Line; New_Line;
```

Once again, 10 is displayed.

T r a p: Although you can declare variables in a function with the same name as a global variable, it’s not a good idea because it can lead to confusion.

Altering Global Variables

Just as you can access a global variable from anywhere in your program, you can alter one from anywhere in your program, too. That’s what I do next, when I call change_global. The key line of the function assigns 10 to the global variable glob.

```
  glob := -10;  -- change global variable glob
```

To show that it worked, I display the variable glob in change_global with:

```
  Put( “In change_global, glob is: ");   Put(Glob); New_Line; New_Line;
```

Then, back in main, I again display glob with:

```
  Put( "In main glob now is: ");  Put(Glob); New_Line; New_Line;
```

Because the global variable glob was changed, 10 is displayed.

Here is the complete code:

```
-- Global Reach: Demonstrates global variables

WITH Text_Io;              USE Text_Io;
WITH Ada.Integer_Text_IO; USE Ada.Integer_Text_IO;

PROCEDURE Global_Reach IS

   Glob : Integer := 10; -- Global Variable

   -- bodies for 3 procedures
   PROCEDURE Access_Global IS
   BEGIN
      Put("In access_global is glob is: ");
      Put(Glob); New_Line; New_Line;
   END;

   PROCEDURE Hide_Global IS
      Glob : Integer := 0;   -- hide global glob with local glob
   BEGIN
      Put("In hide_global, glob is: ");
      Put(Glob); New_Line; New_Line;
   END;

   PROCEDURE Change_Global IS
   BEGIN
      Glob := -10;
      -- Change Global Variable Glob
      Put("In change_global glob is: ");
      Put(Glob); New_Line; New_Line;
   END;

BEGIN  -- main procedure

   Put( "In main glob is: ");
   Put(Glob); New_Line; New_Line;
   Access_Global;  -- call the procedures
   Hide_Global;
   Put( "Back in main glob still is: ");
   Put(Glob); New_Line; New_Line;
   Change_Global;
   Put( "In main glob now is: ");
   Put(Glob); New_Line; New_Line;

END;
```

Wise programmers minimize the use of Global Variables.

Just because you can use them doesn't mean you should. This is a good programming motto. Sometimes things are technically possible, but not a good idea. Using global variables is an example of this. In general, global variables make programs confusing because it can be difficult to keep track of where their values get changed. You should limit your use of global variables as much as possible.

Using Global Constants, on the other hand, is very useful.

Unlike global variables, which can make your programs confusing, global constants—constants that can be accessed from anywhere in your program— can help make programs clearer. You declare a global constant much like you declare a global variable—by declaring it outside of any function. And because you're declaring a constant, you need to use the constant keyword. For example, the following line defines a global constant (assuming the declaration is outside of any function) named MAX_ENEMIES with a value of 10 that can be accessed anywhere in the program.

```
  MAX_ENEMIES : constant := 10;
```

T r a p: Just like with global variables, you can hide a global constant by declaring a local constant with the same name. However, you should avoid this because it can lead to confusion.

How exactly can global constants make game programming code clearer? Well, suppose you're writing an action game in which you want to limit the total number of enemies that can blast the poor player at once. Instead of using a numeric literal everywhere, such as 10, you could define a global constant MAX_ENEMIES that's equal to 10. Then whenever you see that global constant name, you know exactly what it stands for.

This example uses reals:

```
  Speed_of_light : constant := 2.99792458E8  -- m/s
  Max_ship_speed : constant := 0.95 * Speed_of_light
```

As you can see, you can calculate a new constant from one defined previously!

One caveat: You should only use global constants if you need that constant value in more than one part of your program. If you only need a constant value in a specific scope (such as in a single function), use a local constant instead.

6.17 Using Default Arguments

When you write a function in which a parameter frequently gets passed the same value, you can save the caller the effort of constantly specifying this value by using a default argument—a value assigned to a parameter if none is specified.

Here's a concrete example. Suppose you have a function that sets the graphics display. One of your parameters might be a Boolean, fullScreen, which tells the function whether to display the game in full-screen or windowed mode. Now, if you think the function will often be called with true for fullScreen, you could give that parameter a default argument of true, saving the caller the effort of passing true to fullScreen whenever the caller invokes this display-setting function.

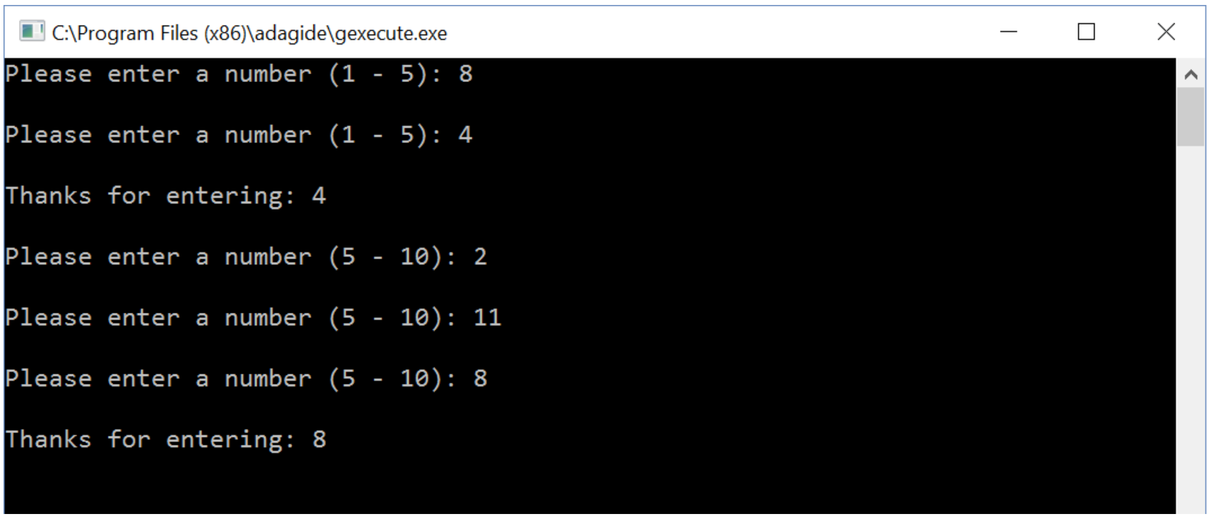
In Ada, many of the library functions have default values. For example, suppose you have defined an integer Number_of_attackers and you display it using integer text IO with

```
Put(Number_of_attackers);     -- This will be displayed with a field wide enough for any integer
                               --  with leading zeros replaces by spaces
Put(Number_of_attackers, 1);   -- This will be displayed with a field width of 1. A bigger field is
                               -- used if the number is too big to fit in that width (here, > 9).
```

Any size field width can be used

6.18 Introducing the Give Me a Number Program

The Give Me a Number program asks the user for two different numbers in two different ranges. The same function is called each time the user is prompted for a number. However, each call to this function uses a different number of arguments because this function has a default argument for the lower limit. This means the caller can omit an argument for the lower limit, and the function will use a default value automatically. Figure 5.5 shows the results of the program.



```
Please enter a number (1 - 5): 8

Please enter a number (1 - 5): 4

Thanks for entering: 4

Please enter a number (5 - 10): 2

Please enter a number (5 - 10): 11

Please enter a number (5 - 10): 8

Thanks for entering: 8
```

Here is the code:

```ada
-- Give Me a Number : Demonstrates default function arguments

WITH Text_Io;                USE Text_Io;
WITH Ada.Integer_Text_IO; USE Ada.Integer_Text_IO;

PROCEDURE GiveMeANumber IS
```

```
   Number : Integer;

FUNCTION AskNumber (High : integer; Low  : Integer := 1) RETURN Integer IS
   Num : Integer;
BEGIN
   LOOP
      Put( "Please enter a number (" );
      Put( Low, 1 ); Put( " - " ); Put( High, 1 ); Put( "): ");
      Get( Num ); Skip_Line; New_Line;
      EXIT WHEN (Num <= High AND Num >= Low);
   END LOOP;  -- repeat until Num in range High -- Low
   RETURN Num;
END;

BEGIN  -- main procedure
   Number := AskNumber(5);  -- get a num in range 1 .. 5
   Put("Thanks for entering: "); Put(Number, 1); New_Line(2);
   Number := AskNumber(10, 5);  -- get a num in range 5 .. 10
   Put("Thanks for entering: "); Put(Number, 1); New_Line(2);
END;
```

Specifying Default Arguments

The function askNumber has two parameters—high and low. You can tell this from the function prototype:

function askNumber(high : integer; low : integer := 1);

Notice that the second parameter, low, looks like it's assigned a value. In a way, it is. The 1 is a default argument meaning that if a value isn't passed to low when the function is called, low is assigned 1. You specify default arguments by using := followed by a value after a parameter name.

T r a p : Once you specify a default argument in a list of parameters, you must specify default arguments for all remaining parameters. so the following specification is valid:

  procedure setDisplay(height, width : integer; depth : integer := 32, fullScreen : boolean := true);

while this one is illegal:

  procedure setDisplay(height, width : integer; depth : integer := 32, fullScreen : boolean);

6.19 Assigning Default Arguments to Parameters

The askNumber function asks the user for a number between an upper and a lower limit. The function keeps asking until the user enters a number within the range, and then it returns the number. I first call the function in the main procedure with:

  number := askNumber(5);

As a result of this code, the parameter high in askNumber() is assigned 5. Because I don't provide any value for the second parameter, low, it gets assigned the default value of 1. This means the function prompts the user for a number between 1 and 5.

Once the user enters a valid number, askNumber returns that value and ends.

Back in the main procedure, the value is assigned to number and displayed.

T r a p: When you are calling a function with default arguments, once you omit an argument, you must omit arguments for all remaining parameters, which must all take their default values. For example, given the specification

```
   procedure setDisplay(height, width : integer; depth : integer := 32, fullScreen : Boolean := true);
```

a valid call to the function would be

```
   setDisplay(1680, 1050);     -- leaves depth = 32 and fullScreen = true
```

while an illegal call would be

```
   setDisplay(1680, 1050, false);    -- Can't set fullScreen without also setting depth
```

This restriction is due to the positional alignment of parameters, so that the arguments in a call to a subprogram line up in order with the specification. This means that the call above to setDisplay tries to pass the argument 1680 to height, 1050 to width and false to depth, but depth must be an integer, not a boolean, so this call results in an error.

Ada, however, also supports named parameters, where parameters can be individually named and called in any order. The names are those given in the subprogram specification, so in the above example, the names are height, width, depth and fullScreen. With this capability, we could write

	height	width	depth	fullScreen
setDisplay(1680, 1050, 16, false);	1680	1050	16	false
setDisplay(height => 1680, width => 1050, fullScreen => false);	1680	1050	32	false
setDisplay(1680, 1050, depth => 16);	1680	1050	16	true
But these calls are illegal:				
setDisplay();	Illegal – no values for height and width			
setDisplay(1680, 1050, false);	Illegal – tries to set depth to false			
setDisplay(fullScreen => false);	Illegal – no values for height and width			

You see that you can override default Arguments. To illustrate this, next I call askNumber again with:

number := askNumber(10, 5);

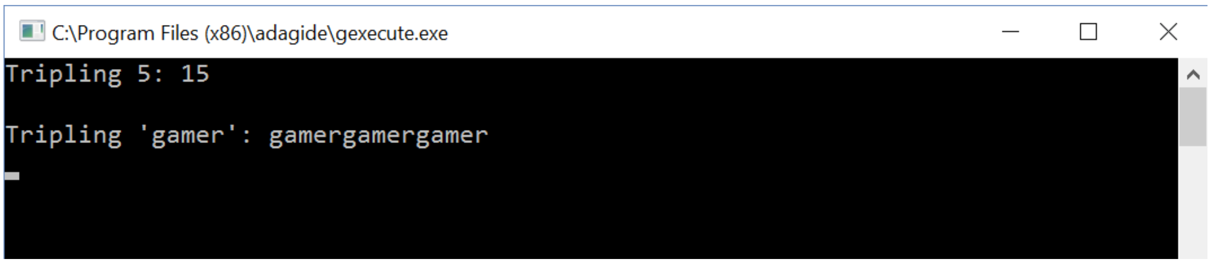
This time I pass a value of 5 for low. This is perfectly fine; you can pass an argument for any parameter with a default argument, and the value you pass will override the default. In this case, it means that low is assigned 5. As a result, the user is prompted for a number between 5 and 10. Once the user enters a valid number, askNumber returns that value and ends. Back in the main procedure, the value is assigned to number and displayed.

6.20 Overloading Functions

You've seen how you must specify a parameter list for any subprogram you write and, if it is a function, a single return type as well. But what if you want a subprogram that's more versatile— one that can accept different sets of arguments? For example, the put procedure has versions that will display text, integers, reals, etc. So, suppose you want to write a function that performs a 3D transformation on a set of vertices that are represented as floats, but you want the function to work with integers as well. Instead of writing two separate functions with two different names, you could use function overloading so that a single function could handle the different parameter lists. This way, you could call one function and pass vertices as either floats or integers.

6.21 Introducing the Triple Program

The Triple program triples the value 5, and the text string “gamer”. The program triples these values using two functions of the same name that have been overloaded to work with an argument of two different types: integer and string. The Ada system is able to decide which to use based on the types of the arguments. If you try to define two functions of the same name and the same argument types, the compiler will tell you it is an error. Figure 5.6 shows a sample run of the program.



```
Tripling 5: 15

Tripling 'gamer': gamergamergamer
```

And here is the code:

```
-- Triple
-- Demonstrates function overloading

WITH Text_Io;                  USE Text_Io;
WITH Ada.Integer_Text_IO;      USE Ada.Integer_Text_IO;

PROCEDURE Triple IS

   FUNCTION Triple (Inp : Integer) RETURN Integer IS
   BEGIN
      RETURN Inp * 3;
   END;

   FUNCTION Triple (Inp : String) RETURN String IS
   BEGIN
      RETURN Inp & Inp & Inp;
   END;

BEGIN
   Put( "Tripling 5: "); Put(Triple(5), 1); New_Line(2);
   Put( "Tripling 'gamer': "); Put(Triple("gamer")); New_Line;
END;
```

As this example shows, to create an overloaded function, you simply need to write multiple functions with the same name and different parameter lists. In the Triple program, I write two definitions for the function triple(), each of which specifies a different type as its single argument. Here are the function specs:

Function triple(Inp : integer) return integer
Function triple(Inp : String) return String

The first takes an integer argument and returns an integer. The second takes a string argument and returns a string.

In each function, you can see that I return triple the value sent. In the first function, I return the integer sent, tripled. In the second function, I return the string sent, repeated three times.

T r a p: To implement subprogram overloading, you need to write multiple subprograms with the same name but with different parameter lists. Notice that I didn't mention anything about return types for function. That's because if you write two function definitions in which only the return type is different, you'll generate a compile error.

You call an overloaded subprogram the same way you call any other subprogram, by using its name with a set of valid arguments. But with overloaded subprograms, the compiler (based on the argument values) determines which of the overloaded subprograms to call.

6.22 Inlining Functions

There's a small performance cost associated with calling a subprogram. Normally, this isn't a big deal because the cost is relatively minor. However, for tiny subprograms (such as one or two lines, like the triple functions above), it's sometimes possible to speed up program performance by inlining them. By inlining a subprogram, you ask the compiler to make a copy of the subprogram everywhere it's called. As a result, program control doesn't have to jump to a different location each time the function is called.

Up to Ada 2005, inlining was specified with an instruction to the compiler, called a pragma. Here is an example. Ada still supports many pragmas to tell the compiler how the code must be built, listed in the LRM Annex L.

```
function cubit(n:integer) return Integer is begin
  return n*n*n; -- A simple function suitable for inlining
end cubit;
pragma inline(cubit);
```

The pragma Inline specifies that a subprogram should be expanded inline everywhere it is called.

With Ada2012, **aspects**, indicated by a **with** statement, are preferred to pragmas. Pragmas still work in Ada2012, but some pragmas are regarded as obsolescent. In Ada 2012 you would write:

```
function cubit(n:integer) return Integer
  with inline is
begin
  return n*n*n; -- A simple function suitable for inlining
end cubit;
```

By flagging the function with inline, you ask the compiler to copy the function directly into the calling code. This saves the overhead of making the function call. That is, program control doesn't have to jump to another part of your code. For small functions, this can result in a performance boost. However, inlining is not a silver bullet for performance. In fact, indiscriminate inlining can lead to worse performance because inlining a function creates extra copies of it, which can dramatically increase memory consumption.

When you inline a function, you really make a request to the compiler, which has the ultimate decision on whether to inline the function. If your compiler thinks that inlining won't boost performance, it won't inline the function.

Calling an inlined function is no different than calling a non-inlined function, so for the cubit function above you could write a line of code like:

```
cubeVolume := cubit(cubeSide);
```

This line of code would calculate the cube of cubeSide and assign the result to cubeVolume.

Assuming that the compiler grants my request for inlining, this code doesn't result in a function call. Instead, the compiler places the code to do the multiplications right at this place in the program. In fact, the compiler does this for all calls to the function.

R e a l World: Although obsessing about performance is a game programmer's favourite hobby, there's a danger in focusing too much on speed. In fact, the approach many developers take is to first get their game programs working well before they tweak for small performance gains. At that point, programmers will profile their code by running a utility (a profiler) that analyses where the game program spends its time. If a programmer sees bottlenecks, he or she might consider hand optimizations such as function inlining.

6.23 Introducing the Mad Lib Game

The Mad Lib game is like the customised story in Chapter 1. It asks for the user's help in creating a story. The user supplies the name of a person, a plural noun, a number, and a verb. The program takes all of this information and uses it to create a personalized story. The difference is that in this version we will use subprograms to build the code. The figure below shows a sample run of the program.

As usual, I start the program with some comments and include the packages I will use.

```
-- Mad-Lib
-- Creates a story based on user input

WITH Text_Io;                  USE Text_Io;
WITH Ada.Strings.Unbounded;    USE Ada.Strings.Unbounded;
WITH Ada.Text_IO.Unbounded_IO; USE Ada.Text_IO.Unbounded_IO;
WITH Ada.Integer_Text_IO;      USE Ada.Integer_Text_IO;
```

Next, I identify the main procedure and declare some variables I will use.

```
PROCEDURE Mad_Lib IS
   Name, Noun, BodyPart, Verb : Unbounded_String;
   Number    : Integer;
```

Now I write the subprograms the main program will call. There are three, two functions and one procedure. The functions are askText which asks the user for text input and askNumber, which asks the player for a number. The procedure is tellStory, which relates the customised story using the inputs from the player

The askText function gets an unbounded string from the player, since we don't know how long a string will be entered. The function is versatile and takes a parameter of type string, which it uses to prompt the player. Because of this, I'm able to call this single function to ask the user for a variety of different pieces of information, including a name, plural noun, body part, and verb.

```
FUNCTION AskText (Prompt : String) RETURN Unbounded_String IS
     Text : Unbounded_String;
 BEGIN
     Put(Prompt); Put(" ");
     Get_line(Text);
     RETURN Text;
 END;
```

Remember that the get_line function with a string input parameter will return with a complete line of text, including white space (such as tabs or spaces) but not the end of line.

The askNumber() function gets an integer from the player. Although I only call it once in the program, it's versatile because it takes a parameter of type string that it uses to prompt the player.

```
FUNCTION AskNumber (Prompt : String)RETURN Integer IS
   Num : Integer;
BEGIN
   Put(Prompt); Put(" ");
   Get(Num); skip_line;
   RETURN Num;
END;
```

Entering non-numerical characters will cause an exception. The Get(Num) call will read a number up to the first white space, so, in case the player enters any other text after the number, I skip the rest of the line.

The tellStory procedure takes all of the information entered by the player and uses it to display a personalized story. Enjoy yourself editing the story and making it as funny as you like.

```
PROCEDURE TellStory ( Name, Noun : Unbounded_String; Number: Integer;
      BodyPart, Verb : Unbounded_String) IS
BEGIN
   New_Line; Put_Line("Here's your story:");
   Put("The famous explorer "); Put(Name);
   Put_Line(" had nearly given up a life-long quest to find");
   Put("The Lost City of "); Put(Noun);
   Put(" when one day, the "); Put(Noun);
   Put_Line(" found the explorer.");
   Put("Surrounded by "); Put(Number, 1);
   Put(" "); Put(Noun); Put(", a tear came to ");
   Put(Name); Put("'s "); Put(BodyPart); Put_Line(".");
   Put("After all this time, the quest was finally over. ");
   Put("And then, the "); Put(Noun); New_Line;
   Put("promptly devoured "); Put(Name); Put_line(". ");
   Put("The moral of the story? Be careful what you ");
   Put(Verb); Put_line(" for.");
   END;
```

The main procedure calls all of the other functions. It calls the function askText to get a name, plural noun, body part, and verb from the player. It calls askNumber to get a number from the player. Finally, it calls tellStory with all of the user-supplied information to generate and display the story.

```
   BEGIN
      Put("Welcome to Mad Lib."); New_Line(2);
      Put("Answer the following questions to help create a new story.");
      New_Line;
      Name := AskText("Please enter a name: ");
      Noun := AskText("Please enter a plural noun: ");
      Number := AskNumber("Please enter a number: ");
      BodyPart := AskText("Please enter a body part: ");
      Verb := AskText("Please enter a verb: ");

      TellStory(Name, Noun, Number, BodyPart, Verb);

   END;
```

In this example I have shown you how to create a program with multiple subprograms, including functions and a procedure.

6.24 A bigger example – Player’s Database

Now let’s look at a bigger example of using subprograms. Many games keep a database of players containing information about them. Remember I set up a record type for a player’s database in section 5.4. Now I will write a program to keep track of the players’ names, phone numbers, spending and payments. I start by making a menu to allow a game administrator to add players, list the players, add to the players’ credit and add to the players’ spending. Here is the menu. Note the use of Get_immediate, which immediately gets the character for the pressed key. This sort of menu can be used for the administration of any group of people, set of things, etc.

```
BEGIN
   Put_line( "Players Database" );
   Membership := 0;
   LOOP
      New_line;
      Put( "Number of members = " ); Put( Membership, 1 ); New_line;
      Put( "A.dd, C.harges, L.ist, P.ayments, Q.uit ? " );
      Get_Immediate( choice ); new_line;
      CASE choice IS
        when 'A' | 'a' => AddRecords;
        when 'C' | 'c' => AddToCharges;
        when 'P' | 'p' => MakePayment;
        when 'L' | 'l' => ListRecords;
        when 'Q' | 'q' => Exit;
        when others    => Put_line( "Invalid choice. Please try again. " );
      END case;
   END LOOP;
END;
```

The program needs a set of records to hold the data for each player. I create this by setting up an array of records:

```
   Type Member is
      RECORD
```

```
         Name      : String(1..30) := (1 .. 30 => ' ');
         Phone     : String(1..12) := (1 .. 12 => ' ');
         Charges   : Float;
         Payments  : Float;
      END RECORD;

   Players : ARRAY( 1 .. Players ) OF Member; -- array of member data.
```

Next will need a set of procedures, which we have already named in the menu:

```
AddRecords;    -- add new players
AddToCharges; -- add expenditure to a player's account
MakePayment;  -- add a payment to a player's credit
ListRecords;  -- list players.
```

We have not shown any parameters for these procedures. If we need to add any, we can do so when we design the procedures. Let's now consider these procedures. We will take ListRecords first. It will list each player's name, phone number, charges, payments and balance. To make it easier to keep track of the players, we will keep the number of players in a variable, Membership. We can then run a loop from 1 to Membership to list the players. Ada uses the '.' notation to select each item for each player, as can be seen in the example code below:

```
PROCEDURE ListRecords is
BEGIN
   Put( "#   Name                      Phone   Charges Payments Balance" );
   FOR number in 1 .. Membership LOOP
      Put( number, 1); Put( " : " ); Put( Players(number).name );
      Put( " " ); Put( Players(number).Phone );
      Put( Players( number ).Charges, fore => 5, aft => 2, exp => 0 );
      Put( Players( number ).Payments, fore => 5, aft => 2, exp => 0 );
      Put( Players( number ).Charges
         - Players( number ).Payments, fore => 5, aft => 2, exp => 0 );
      New_line;
   END LOOP;
END ListRecords;
```

This will produce a list of players that looks like:

```
#   Name                     Phone     Charges Payments Balance
1 : Fred Jones              36471259   121.96    93.50   28.46
2 : Jack Diamond            54231183  6432.17  6500.00  -67.83
```

Etc.

Now consider the addRecords procedure. We ask for the name of the new player, his/her phone number, set the charges and payments to zero and increase the membership by 1.

```
PROCEDURE AddRecords IS
   last : integer;
BEGIN
   IF Membership = MaxMembers
      THEN Put_line( "Membership is full" );
      ELSE
         Put_line( "Add new record" );
         Membership := Membership + 1;
         Put( "Name  : " ); Get_line( Players( Membership ).Name );
         Put( "Phone : " ); Get_line( Players( Membership ).Phone );
         Players( Membership ).Charges := 0.0;
         Players( Membership ).Payments := 0.0;
         Put_line( "Thank you for joining." );
         Put( "Your membership number is " ); Put(Membership);
         New_Line;
      END IF;
END AddRecords;
```

A complete version of this program, PlayersDatabase is in the examples. It has a many deficiencies, of course, being a relatively simple example. For instance, there is no way to delete a player who leaves. As an exercise, try writing some code to deal with this. Consider what happens to the deleted record.

A greater difficulty is that when we quit the program, all the data that was entered is lost. This is because the variables, records and array (the data structures) in the program are *volatile,* that is, they are discarded when the program ends and are reset to their initial values when the program is run the next time. To avoid this, we need to save the data in persistent storage, that is, in a disk file, which preserves the data, and then read it in again from the file when the program restarts. We will see how to do this in chapter 8.

We will now look at another concept associated with subprograms, recursion.

6.25 Recursion

Many mathematical functions are defined recursively, that is, in terms of themselves, so they form a series, and many functions that manipulate data structures use recursion. Games use recursion to search the game board or to search for a good strategy.

Recursion occurs when a program is able to call itself. This ability is often called re-entrancy. In Ada, all subprograms are intrinsically recursive, provided they do not use any variables defined outside of the subprogram.

Let's write a simple function to use recursion to calculate the number n! (factorial) for positive numbers. It is defined as 1! = 1, n! = n * (n-1)! You can see that the function is defined in terms of itself and it has an ending condition, so it is therefore recursive.

```
FUNCTION Fact (N : Natural) RETURN Natural IS
  BEGIN
    IF N = 0 THEN RETURN 0;
    ELSIF N = 1 THEN RETURN 1;
    ELSE RETURN N * Fact(N-1);
    END IF;
  END;
```

The recursive function is very simple and closely follows the mathematical definition. A complete program to test it is in Fact.adb. The size of the result grows very quickly. Try running it for various values of N.

Recursion can, if not used with care, eat up CPU resources. Consider the Fibonacci calculation we looked at previously. Mathematically, it is defined as F(0) = 0, F(1) = 1, F(N) = F(N-1) + F(N-2). Clearly, the function is defined in terms of itself, together with an ending condition, and it is therefore recursive. We can write this in Ada as

```
Function Fib(N : natural) return natural is
BEGIN
   IF N = 0 THEN RETURN 0;
   ELSIF N = 1 THEN RETURN 1;
   ELSE RETURN Fib(N-1) + Fib(N-2);
   END IF;
END;
```

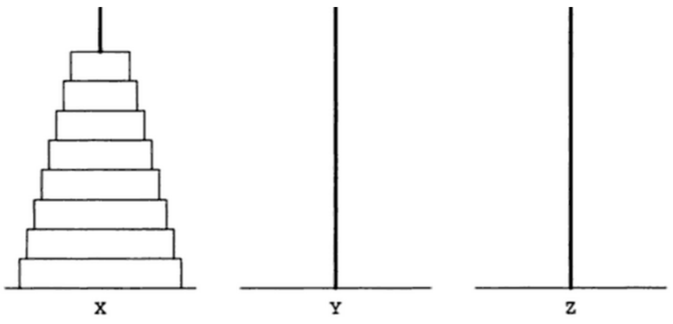
Note the simplicity of the recursive form of the function. A complete program to test it is in FibNRec.adb. Note we have chosen to use the predefined natural integers for the function parameter N, which have a range 0 .. Integer'Last. A program that uses this function then cannot call it with a negative integer and a program that tries to call it with an integer parameter will not compile. This is in line with Ada's philosophy of strong typing, one of the defences the language provides against programming errors.

Unfortunately, the simplicity of this program brings a disadvantage. Load and run FibNRec and enter 45 for N. You will see that it takes a noticeable amount of time to do the calculation. This is because of the double call to itself in the line `RETURN Fib(N-1) + Fib(N-2);` which results in the algorithm taking of the order of 2^N amount of time to complete, which for even a small value for N is a big number. There are neat algorithms that fix this problem, but they are beyond our present scope. An example of one, however, is included as FibN.adb, which we looked at earlierand which does the calculation iteratively. In general double (or multiple) recursion is not a good idea!

Recursive calculations can always be done iteratively (i.e., with loops) for greater efficiency, but the simplicity of the recursive solutions means that there is less chance of making programming errors.

Let's now look at a classic recursive puzzle, the Towers of Hanoi.

6.26 Towers of Hanoi Puzzle



Many games revolve around puzzles and many puzzles are recursive in nature. Let's look at a classic example of a recursive puzzle, the Towers of Hanoi. The Tower of Hanoi puzzle was invented by the French mathematician Edouard Lucas in 1883. He was inspired by a legend that tells of a Hindu temple where the puzzle was presented to young priests. At the beginning of time, the priests were given three poles and a stack of 64 gold disks, each disk a little smaller than the one beneath it. Their assignment was to transfer all 64 disks from one of the three poles to another, with two important constraints. They could only move one disk at a time, and they could never place a larger disk on top of a smaller one. The priests worked very efficiently, day and night, moving one disk every second. When they finished their work, the legend said, the temple would crumble into dust and the world would vanish.

We can define this problem recursively as follows:

```
Whle n > 0
   Move n-1 disks from X to Z using Y
   Move nth disk from X to Y
   Move n-1 disks from Z to Y using X
```

This is a very simple solution that demonstrates again the great power of recursion, where use of recursion easily solves a problem that is difficult to solve iteratively.

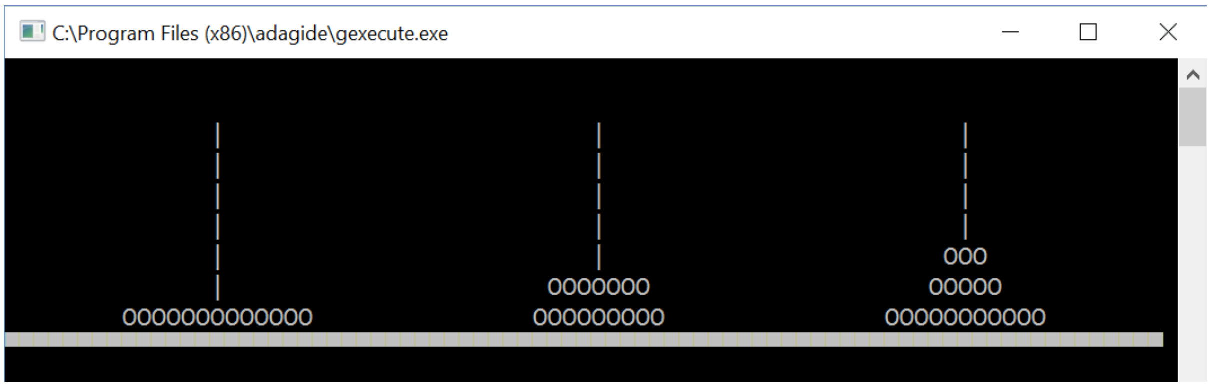
A program that generates the set of moves for n disks is given below:

```
WITH Text_Io; USE Text_Io;
WITH Ada.Integer_Text_IO; USE Ada.Integer_Text_IO;

PROCEDURE Hanoi IS
   NumDisks : Natural;
   PROCEDURE Han(N: Natural; X,Y,Z:Character) IS
   BEGIN
      IF N > 0 THEN
         Han(N-1, X,Z,Y);
         Put("Move disk "); Put(N, 1); Put(" from ");
         Put(X); Put(" to "); Put(Y);
         New_Line;
         Han(N-1, Z,Y,X);
      END IF;
   END Han;

BEGIN
   Put("Number of disks to move: ");
   Get(NumDisks);
   Han(Numdisks, 'X', 'Y', 'Z');
END;
```

It is interesting to run this program and examine the series of moves. The number of moves increases very rapidly with the number of disks ($O(2^N)$), so even for relatively small values of N, there will be too many moves to decipher.

To watch an animated version of the towers of Hanoi, slowed down to see the moves, see `HanoiAnim.adb`. This program uses the goto_XY procedure from Gerry van Dijk's NT_Console interface to the Windows console screen to position the cursor on the screen in order to erase and draw the disks. You are invited to explore the code.

Here is a screen shot of the program in the process of moving 6 disks:


C:\Program Files (x86)\adagide\gexecute.exe

```
               |                         |                        |
               |                         |                        |
               |                         |                        |
               |                         |                        |
               |                         |                       000
               |                      0000000                   00000
         0000000000000               000000000               00000000000
```

The program draws the disks as a set of ‘O’s on a baseline, with vertical lines for the towers. The recursive procedure Han is the same as before, with the text output removed and two new procedures added, DeleteDisk and AddDisk:

Original Han procedure

```
   PROCEDURE Han(N: Natural;
       X,Y,Z:Character) IS
   BEGIN
      IF N/= 0 THEN
         Han(N-1, X,Z,Y);
         Put("Move disk ");
         Put(N,1);
         Put(" from ");
         Put(X); Put(" to ");
         Put(Y);
         New_Line;
         Han(N-1, Z,Y,X);
      END IF;
   END Han;
```

Modified Han procedure

```
   PROCEDURE Han (N : Natural;
         X,Y,Z : Character) IS
   BEGIN
      IF N/= 0 THEN
         Han(N-1, X,Z,Y);
         DeleteDisk(X);
         AddDisk(N, Y);
         Han(N-1, Z,Y,X);
      END IF;
   END Han;
```

Without the text output the modified procedure is quite simple. But now I need to add code to draw the original stack of disks, drawDisks, then, for each move, DeleteDisk from a column and add that disk to another column.

I have drawn each disk as 2*N + 1 ‘O’s, so the top disk (disk 1) is drawn as 000, the second as 00000, etc. The towers are spaced at columns 15, 41 and 66, evenly across the screen and shown as vertical bars. This allows for up to 12 disks, the widest being 25 ‘O’s wide.

```
-- Towers of Hanoi simulation
-- Limited to max of 12 disks due to screen size

WITH Text_Io;                USE Text_Io;
WITH Ada.Integer_Text_IO;    USE Ada.Integer_Text_IO;
WITH NT_Console;             USE NT_Console;

PROCEDURE HanoiAnim IS
   NumDisks     : Natural RANGE 1 .. 12;
   Xd, Yd, Zd   : Natural := 0;
   MovesPerSec  : Integer;
   MoveDelay    : Duration;

   PROCEDURE DeleteDisk (X : Character) IS
      Col      : Natural;
      ThisDisk : Natural;
   BEGIN
      CASE X IS
         WHEN 'X' =>
            BEGIN
               Col := 14;
               ThisDisk := Xd;
               Xd := Xd - 1;
            END;
         WHEN 'Y' =>
            BEGIN
               Col := 40;
               ThisDisk := Yd;
               Yd := Yd - 1;
            END;
         WHEN 'Z' =>
            BEGIN
               Col := 65;
               ThisDisk := Zd;
               Zd := Zd - 1;
            END;
         WHEN OTHERS =>
            NULL;
      END CASE;
      IF MovesPerSec /= 0 THEN
         DELAY MoveDelay;
      END IF;
      Goto_Xy(Col-Numdisks,Numdisks-ThisDisk+3);
      FOR I IN 1..Numdisks LOOP
         Put(" ");
      END LOOP;
      Put("|");
      FOR I IN 1..Numdisks LOOP
         Put(" ");
      END LOOP;
   END;

   PROCEDURE AddDisk (
         N : Natural;
         X : Character) IS
      Col      : Natural;
      ThisDisk : Natural;
   BEGIN
      CASE X IS
         WHEN 'X' =>
            BEGIN
               Col := 14;
               ThisDisk := Xd;
               Xd := Xd + 1;
            END;
         WHEN 'Y' =>
            BEGIN
               Col := 40;
               ThisDisk := Yd;
               Yd := Yd + 1;
            END;
         WHEN 'Z' =>
            BEGIN
               Col := 65;
               ThisDisk := Zd;
```

```
               Zd := Zd + 1;
            END;
         WHEN OTHERS =>
            NULL;
      END CASE;
      Goto_Xy(Col-N,Numdisks-ThisDisk+2);
      FOR I IN 1..2*N+1 LOOP -- draw disk
         Put("O");
      END LOOP;
   END;

   PROCEDURE Han (
         N : Natural;
         X, Y, Z : Character) IS
   BEGIN
      IF N/= 0 THEN
         Han(N-1, X,Z,Y);
         DeleteDisk(X);
         AddDisk(N, Y);
         Han(N-1, Z,Y,X);
      END IF;
   END Han;

   PROCEDURE DrawDisks (
         N : Integer) IS
   BEGIN
      Set_Col(15);
      Put('|');
      Set_Col(41);
      Put('|');
      Set_Col(66);
      Put('|');
      New_Line;
      FOR I IN 1..N LOOP
         Set_Col(Positive_Count(15-I));
         FOR J IN 1..2*I+1 LOOP
            Put('O');
         END LOOP;
         Set_Col(41);
         Put('|');
         Set_Col(66);
         Put('|');
         New_Line;
      END LOOP;
      FOR I IN 1 .. 79 LOOP
         Put(Character'Val(223));
      END LOOP;
   END;

BEGIN
   Put_Line("Towers of Hanoi");
   New_Line;
   LOOP
      BEGIN
         Put("Number of disks to move (1 .. 12): ");
         Get(NumDisks);
         EXIT;
      EXCEPTION
         WHEN OTHERS =>
            Put_Line("Try again");
      END;
   END LOOP;
   Put("How fast must I move the disks (moves per second, 0 = max): ");
   Get(MovesPerSec);
   IF MovesPerSec /= 0 THEN
      MoveDelay := Duration(1.0/Float(MovesPerSec));
   END IF;
   Clear_Screen;
   Goto_Xy(3,1);
   Xd := Numdisks;
   Drawdisks(Numdisks);
   Han(Numdisks, 'X', 'Y', 'Z');
END;
```

6.27 Developing good software

In the examples above, we have broken up our programs into manageable chunks by using procedures and functions. This helps a great deal in the design of more complex games and in fact all software. But how do we know we have a good design? A good start is to put your ideas down on paper, like an architect planning a building, using pseudocode, before you write your actual code. Then you can review your design before committing it to code.

The next major technique is to use the great power of the Ada library packages, which contain many useful subprograms, rather than writing your own. We have already used many Ada library subprograms and you need to explore more in order to get a decent grasp of the Ada libraries. By simply “withing” the packages and calling appropriate subprograms in the body, you can reduce code size dramatically.

In addition, the subprograms in the standard library packages have tested thoroughly, written by experts and carefully optimized. They also handle errors well, so you can rely on them not to crash your program. For example, the Ada Language Reference Manual (LRM) tells you exactly what will happen is an error occurs in calling a subprogram. For example, in the Numerics package, if the sqrt function is called with a parameter with a negative value, the exception Numerics.Argument_Error is raised, signalling a parameter value outside the allowed range of the sqrt function.

Subprograms should be designed to do a single thing well. If, however, a function is very small, it might be inlined. On the other hand, if a subprogram gets too big, it might be a good idea to break it up into a set of smaller subprograms, which can be grouped into a package, as will be discussed in the next section.

So the first thing to do in designing a subprogram is to make sure you understand very clearly exactly what it is supposed to do. Put down on paper (not on the computer) what you think your program will do, think about it, try it out mentally, write some rough pseudocode and refine it before writing a line of actual code.

Once you are confident of what you want the subprogram to do, write down its parameters. This is the data you will be sending to the subprogram. This data will be processed by the subprogram using variables and data structures, so your next step is to identify these. Write down any variables you may need along with any structures you need, such as arrays, vectors, etc. For example, in board games, we need to consider how we will represent the board and in card games, the cards dealt to the players. We will look at examples of these in later sections

6.28 Pre- and postconditions

You also need to identify any other conditions that must apply at the start of the subprogram. These are called the preconditions of the subprogram. For example, if your subprogram reads the value of a global variable, you may need it to be in a certain range. Perhaps in an adventure game, the hero carries varies items, such as a special key, magic potion, etc., you may require that she is carrying at least one item when the subprogram is called.

Frequently, preconditions can be enforced by choosing the types of the parameters correctly. For example, if you were to write a subprogram to list the weapons in the possession of an adventurer, the object passed to the subprogram should be of a user-defined type something like weaponsList.

Postconditions are similarly important. They are what must be true at the end of the subprogram. They can also be enforced to some extent by choosing the types of variables for returning results, to make sure they do not return bad results. For example, our adventurer may only be able to carry up to 3 weapons, and if he attempts to pick up more, this should be prevented by a check such as raising an exception.

Pre- and postconditions are specified in Ada using a “With” clause in the subprogram header and are examples of Aspects. A full treatment of Aspects is beyond the scope of this short book. You are encouraged to look at the Ada 2012 Rationale for more information.

For now, let’s look at a few examples. Suppose you actually wanted to write your own square root function and you realise that the parameter X must not be negative. You could write:

```
Function sqrt(x : float) return float is
With
   pre => x >= 0.0;  -- x must be greater than or equal to zero
begin
   -- and so on
```

Postconditions can be used to check that a result of a calculation is within a reasonable range. Suppose we have a function that computes a result that must be in the range 3.0 .. 5.0:

```
Function MyCalc(p : float) return float is
With
   Post => MyCalc is in 3.0 .. 5.0;
```

Postconditions are very useful in checking that subprograms have not gone haywire and are returning good results.

6.29 Testing

Once you are happy with you design, it is time to start coding, and as you code the subprogram you should test it as you go. Testing individual subprograms is vital and will need a short program, called a driver, to set up data and call the subprogram to check it is doing what you plan. Wring all the code and then testing the complete program is a major error novice programmers make, because the complete programs with several subprograms is large and complex, which makes testing and debugging very difficult. It’s much better to code a little, test a little, code a little more, test some more, and so on.

Very often in testing you need to know the values of the variables as the program executes. In a good IDE such as AdaGIDE, it is possible to pause the program and check the values of each variable in the subprogram by executing it line by line and making sure the execution proceeds as you expect. It pays to invest time in becoming skilled at using the IDE’s debugging facilities!

7. Packages

As you have seen, Ada programs often use subprograms, which are a kind of program unit. Ada provides several ways to organise programs into program units. They can be:

- subprograms, which can be procedures or functions and which define operations to be executed.
- packages, which define collections of entities, which may be subprograms or other collections of resources. Packages are the main grouping mechanism in Ada and are heavily used in Object Oriented Programming in Ch 10.
- Tasks, protected units and generics. These three are more advanced topics, so will be covered later

Most Ada programs are basically a set of a large number of packages, with one procedure used as the ``main'' procedure to start the Ada program.

Packages help manage large scale programs. As the Hanoi example shows, the complexity and size of a program can grow very quickly additional features are added. To keep things under control, Ada supports the concept of programming with new and existing modules of software. The idea is to create modules whose implementation integrity is preserved in the presence of reuse. This helps programmers to construct large programs from multiple modules.

The Ada system itself makes heavy use of packages, providing packages for:

- Text, integer, real and enumerated IO
- Character and string handling
- Containers
- Numerical functions and random number generation

and lots more.

Programmers are encouraged to design and build their own packages, then reuse them frequently to reduce the amount of new code they write. Commercial software companies develop lots of packages and aim for a very high level of code reuse to reduce the cost of software development.

Specifications and bodies

All modules, or program units, in Ada are defined by their specification (spec) and their body. The specification, defines the interface of the module to clients while the body defines its implementation. Separating the two protects the integrity of the module implementation. Note that some authors talk of declarations or prototypes rather than specifications.

The body of a module may change without recompiling any clients but any change to the module specs will cause all clients (units that call on the module) to be recompiled. This approach allows teams of developers to approach a large scale development by writing and compiling the set of specs that will make up the program to ensure they will all interface properly.

These separate parts of program units are usually stored in separate files with the specs having a file extension typically of ads while the bodies have file extension adb. This explicit distinction between declaration and body allows a program to be designed, written, and tested as a set of largely independent software components. The specs are written and compiled and when they all compile correctly, the developers go off and write the corresponding bodies

Separate specifications are not required for subprograms (procedures and functions). If a subprogram has a body but no specification, the body of a subprogram serve as its own spec. This makes writing simple programs easier - technically, that simple program is a procedure body that defines its own spec.

Packages are used for all sorts of collections of programming resources. Packages are modules which encapsulate related set of resources to service clients. The resources that a package can provide are all those that can be declared in Ada. To access the resources in the package, the programmer uses the keyword "with".

Packages are used for many things. Here are some:

- Collections of Declarations
- Collections of Subprograms
- Abstract Data Types
- Abstract State Machines
- Generics (program templates)

In games, packages are used to define collections of things. Here is a package that consists only of a spec. There is no need for a body as it contains only declarations:

```
Package METRIC_EARTH_CONSTANTS is
      EQUATOR_RADIUS : constant := 6_378.145;
      -- units are km
      GRAVITATION : constant := 3.968_012e5;
      -- units are km**3/sec**2
      SPEED_THRO_SPACE := 7.905_368_28;
      -- units are km/sec
end METRIC_EARTH_CONSTANTS;
```

A client accesses the package with the statement:

```
With METRIC_EARTH_CONSTANTS;
```

Now let's look at the specification and body for the subprograms used by the mad_lib program in the preceding chapter.

```
-- Mad-Lib package spec
-- Subprograms for mad_lib, a story based on user input

Package MadLibPak is

function askText(prompt : string) return string;
function askNumber(prompt : string) return integer;
procedure tellStory(name, noun : string; number : integer;
                    bodyPart, verb : string);
end;
```

Notice that the package spec lists only the subprogram specs, with no code. This spec, however, should be saved in a file. The file name must match the package name, MadLibPak, and the file extension must be .ads (for Ada spec). Once saved, it should be compiled to check for errors, although of course it won't run as there in no code.

The body of MadLibPak will contain the executable code for the subprograms, that can then be called by the client program.

```
-- Mad-Lib
-- Creates a story based on user input

WITH Text_Io;              USE Text_Io;
WITH Ada.Integer_Text_IO; USE Ada.Integer_Text_IO;

PACKAGE BODY MadLibPak IS

   FUNCTION AskText (Prompt : String) RETURN String IS
   BEGIN
      Put(Prompt);
      RETURN Get_Line;
   END;

   FUNCTION AskNumber (Prompt : String) RETURN Integer IS
      Num : Integer;
   BEGIN
      Put(Prompt);
      Get(Num);
      skip_line;
      RETURN Num;
   END;

   PROCEDURE TellStory (Name, Noun : String; Number : Integer;
                        BodyPart, Verb : String) IS
   BEGIN
      New_Line; Put( "Here's your story:"); New_Line(2);
      Put( "The famous explorer "); Put( Name);
      Put( " had nearly given up a life-long quest to find");
      New_Line;
      Put( "The Lost City of "); Put( Noun);
      Put( " when one day, the "); Put( Noun);
      Put( " found the explorer."); New_Line;
      Put( "Surrounded by "); Put( Number, 2);
      Put( " " ); Put(Noun);
      Put( ", a tear came to "); Put( Name );
      Put ( "'s "); Put( BodyPart ); Put_line( ".");
      Put( "After all this time, the quest was finally over. ");
      Put( "And then, the "); Put( Noun ); new_line;
      Put( "promptly devoured "); Put( Name ); Put( ". ");
      Put( "The moral of the story? Be careful what you ");
      Put( Verb); Put( " for.");
   END;
END;
```

The package body starts with the line containing the keyword BODY:

```
PACKAGE BODY MadLibPak IS
```

followed by the code for the subprograms, and will be stored in a file MadLibPak.adb (for Ada body). It should be compiled to check for errors and for compliance with the specification. It is not yet executable as there is no main procedure yet.

I can now write the main procedure which calls the subprograms contained in package. To access the package, I use With madlibpak, which gives access to it, followed by Use madlibpak to make all the resources in the package visible.
The main procedure is now very simple. It simply calls the subprograms in the package, and they do all the work.

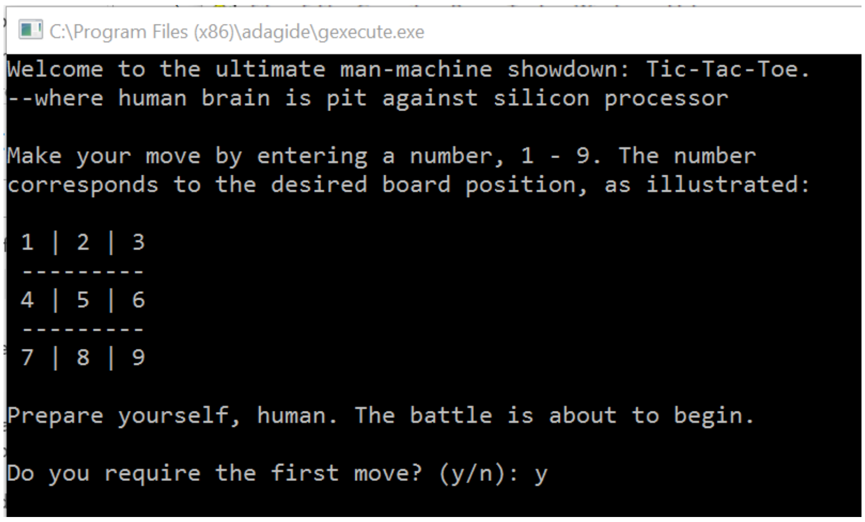
```
-- Mad-Lib
-- Creates a story based on user input

WITH Text_Io;   USE Text_Io;
with madlibpak; use madlibpak;

PROCEDURE Mad_Lib_Main IS
BEGIN
   Put ("Welcome to Mad Lib.");
   New_Line;
   New_Line;
   Put("Answer the following questions to help create a new story.");
   new_line;
      DECLARE
         Name : string := AskText("Please enter a name: ");
         Noun : string := AskText("Please enter a plural noun: ");
         Number : integer := AskNumber("Please enter a number: ");
         BodyPart : string := AskText("Please enter a body part: ");
         Verb : string := AskText("Please enter a verb: ");
      BEGIN
         TellStory(Name, Noun, Number, BodyPart, Verb);
      END;
   END;
```

At this point, packages do not seem to do much for us, but as programs get bigger, the ability to define separate packages and to compile and test them separately becomes very useful. The package itself should be compiled to check for errors, but it won’t run by itself. It needs a driver procedure to call the subprograms in the package.

8. Building a larger game

Let’s now tackle a larger project, building a Tic-Tac-Toe (noughts and crosses) game. We will use quite a bit of what we have learned so far and will also apply the design principles we have talked about.

In this game project, you’ll learn how to create a computer opponent using a dash of AI (Artificial Intelligence). In the game, the player and computer square off in a high-stakes, man-versus-machine showdown of Tic-Tac-Toe. The computer plays a formidable (although not perfect) game and comes with enough attitude to make any match fun. Figure 8.1 shows the start of a match.



```
Welcome to the ultimate man-machine showdown: Tic-Tac-Toe.
--where human brain is pit against silicon processor

Make your move by entering a number, 1 - 9. The number
corresponds to the desired board position, as illustrated:

 1 | 2 | 3
 ---------
 4 | 5 | 6
 ---------
 7 | 8 | 9

Prepare yourself, human. The battle is about to begin.

Do you require the first move? (y/n): y
```

8.1 Planning the Game

This game is your most ambitious project yet. You certainly have all the skills you need to create it, but I’m going to go through a longer planning section to help you get the big picture and understand how to create a larger program. Remember, the most important part of programming is planning the program. Without a roadmap, you’ll never get to where you want to go (or it’ll take you a lot longer as you travel the scenic route).

Real World: Game designers work countless hours on concept papers, design documents, and prototypes before programmers write any game code. Once the design work is done, the programmers start their work— more planning. It’s only after programmers write their own technical designs that they then begin coding in earnest. The moral of this story? Plan. It’s easier to scrap a blueprint than a 50-story building.

8.2 Writing the Pseudocode

It’s back to your favourite language that’s not really a language—pseudocode. We will use this to plan our program. Because I’ll be using subprograms for most of the program, I can afford to think about the code at a pretty abstract level. Each line of pseudocode should feel like one subprogram call. Later, all I’ll have to do is write the subprograms that the plan implies. Then I’ll bundle them all together into a package, so the main program will be pretty simple. Here’s the pseudocode:

```
Create an empty Tic-Tac-Toe board
```

```
Display the game instructions
Ask who goes first
Display the board
While nobody's won and it's not a tie
    If it's the human's turn
        Ask for the human's move
        Update the board with the human's move
    Otherwise
        Calculate the computer's move
        Update the board with the computer's move
Display the board
Switch turns
Congratulate the winner or declare a tie
Ask if the player wants to play again; if yes, start again.
```

8.3 Representing the Data

All right, I've got a good plan, but it is pretty abstract and talks about throwing around different elements that aren't really defined in my mind yet. I see the idea of making a move as placing a piece on a game board. But how exactly am I going to represent the game board? Or a piece? Or a move? Since I'm going to display the game board on the screen, why not just represent a piece as a single character—an X or an O? An empty piece could just be a space. Therefore, the board itself could be a vector of characters. There are nine squares on a Tic-Tac-Toe board, so the vector should have nine elements. Each square on the board will correspond to an element in the vector. The game board below shows what I mean.

1	2	3
4	5	6
7	8	9

Each square or position on the board is represented by a number,1-9. That means the vector will have nine elements, giving it position numbers 1-9. Because each move indicates a square where a piece should be placed, a move is also just a number, 1-9. That means a move could be represented as an integer.

The side the player and computer play could also be represented by a character— either an 'X' or an 'O', just like a game piece. A variable to represent the side of the current turn would also be a char, either an 'X' or an 'O'.

8.4 Creating a List of Functions

The pseudocode inspires the different subprograms I'll need. I created a list of them, thinking about what each will do, what parameters they'll have, and what values they'll return. Looking at the pseudocode, I see we need

a display board procedure,
a function to ask if the player wants to go first,

a function to get the player’s move (a number in the range 1-9),
a function to work out the computer’s move,
a function to decide if there is a winner,
a function to check if a move is legal (ie the square is not occupied).
and so on

The list below shows the results of my efforts. Creating this list of subprograms maps out the architecture of the game software and, just like an architect designing a building, it may need several versions before you are happy with the basic plan.

8.5 Setting Up the Program

The code for this program is available as usual. I’ll go over the code here, section by section to explain what I am doing. The first thing I will do is set up the main procedure, called Tic_Tac, which will be very simple, with all the subprograms that do the work in the supporting package, which I will call Tic_Tac_Pak.

Here is the main program:

```
WITH Tic_Tac_Pak; USE Tic_Tac_Pak;

PROCEDURE Tic_Tac IS
BEGIN
   Instructions;
   Play;
END;
```

All it does is ‘with’ Tic_Tac_Pak, then call two procedures, Instructions to give the player some instructions on how to play, and Play, to actually play the game. So now I need to create the spec for the Tic_Tac_Pak package, in order to capture my approach to the design of the game software.

But before I do that, I must decide how I want to represent the game board. I already decided it will be a vector of 9 characters, but should we use an Array or a Container? This decision is based on the Arrays being well designed for vectors with fixed lengths, whereas the Container is best when the vector’s size has to vary dynamically (i.e. while the program runs). So here we will use an array. Just for fun, I created both solutions and you can look at the solution using a container if you are interested. Here is my spec for the version using an array:

```
PACKAGE Tic_Tac_Pak IS  -- this is the specification
   NUM_SQUARES : CONSTANT Integer := 9;
   -- define some constants as shorthand
   X       : CONSTANT Character := 'X';
   O       : CONSTANT Character := 'O';
   EMPTY   : CONSTANT Character := ' ';
   TIE     : CONSTANT Character := 'T';
   NO_ONE  : CONSTANT Character := 'N';
   -- define the board
   TYPE Board_Type IS ARRAY (Integer RANGE 1 .. 9) OF Character;
   Board : Board_Type := (others => EMPTY);  -- initialised to empty
   Move  : Integer;
   Human, Computer, Turn : Character;

   PROCEDURE Instructions;    -- Displays the game instructions.
   PROCEDURE Play;            -- Play the game
   FUNCTION AskYesNo (Question : String)RETURN Character;  -- displays the
        -- Question, asks for a y/n reply, returns y or n
   FUNCTION AskNumber (Question : String; High : Integer; Low : Integer
       := 1) RETURN Integer;   -- displays the Question, asks for a number
        -- in the range Low - High, returns that number
   FUNCTION HumanPiece RETURN Character;  -- Determines the human’s piece.
      -- Returns either an ’X’ or an ’O’.
   FUNCTION Opponent ( Piece : Character) RETURN Character; -- swaps pieces
   PROCEDURE DisplayBoard ( Board : Board_Type);  -- displays current board
   FUNCTION Winner ( Board : Board_Type) RETURN Character;  -- Checks the
      -- board for a winner. Returns X, O, T (tie), or ’N’ (no winner yet).
   FUNCTION IsLegal ( Board : Board_Type; Move : Integer) RETURN Boolean
      with inline; -- Checks if the move is legal (to an unoccupied square)
   FUNCTION HumanMove ( Board :  Board_Type) RETURN Integer; -- Gets the
      -- human’s move and checks if it’s legal
   FUNCTION ComputerMove ( Board : Board_Type;
      Computer : Character) RETURN Integer;   -- Calculates computer’s move
   PROCEDURE AnnounceWinner ( Winner, Computer, Human : Character);
      -- Announce and congratulate the winner
END Tic_Tac_Pak;
```

I defined these subprograms pretty much directly from the pseudocode and by thinking about the play sequence. Some of them, AskYesNo and AskNumber, I have already used in MadLib. Instructions will simply display a list of instructions to the player. The rest will need considerable work to fill in the details of the body of the code. I will go through them in detail, one by one. Here is the start of the package body for Tic_Tac_Pak.adb, along with the first procedure, Instructions, which displays a set of instructions and gives a little attitude to the player:

```
WITH Text_Io; USE Text_Io;
WITH Ada.Integer_Text_IO; USE Ada.Integer_Text_IO;

PACKAGE BODY Tic_Tac_Pak_2 IS  -- this is the body

   PROCEDURE Instructions IS
   BEGIN
      Put_Line( "Welcome to the ultimate man-machine showdown: Tic-Tac-Toe.");
      Put_Line( "--where human brain is pit against silicon processor");
      New_Line;
      Put_Line( "Make your move by entering a number, 1 - 9. The number");
      Put_Line( "corresponds to the desired board position, as illustrated:");
      New_Line;
      Put_Line( " 1 | 2 | 3");
      Put_Line( " --------- ");
      Put_Line( " 4 | 5 | 6");
      Put_Line( " --------- ");
      Put_Line( " 7 | 8 | 9");
      New_Line;
      Put_Line( "Prepare yourself, human. The battle is about to begin.");
      New_Line;
   END;
```

The next subprogram in the spec is Play, which starts with an outer loop so that, at the end of a game, the human player will be asked if he/she wishes to play again. The game loop then follows the structure of the pseudocode:

```
   PROCEDURE Play IS
   BEGIN
      PlayAgain: LOOP
         Human := HumanPiece;
         Computer := Opponent(Human);
         Turn := X;                 -- Player who goes first uses X
         Board := (OTHERS => EMPTY);
         DisplayBoard(Board);
         WHILE (Winner(Board) = NO_ONE) LOOP
            IF (Turn = Human) THEN
               BEGIN
                  Move := HumanMove(Board);
                  Board(Move) := Human;
               END;
            ELSE
               BEGIN
                  Move := ComputerMove(Board, Computer);
                  Board(Move) := Computer;
               END;
            END IF;
            DisplayBoard(Board);
            Turn := Opponent(Turn);
         END LOOP;
         AnnounceWinner(Winner(Board), Computer, Human);
         EXIT PlayAgain WHEN (AskYesNo("Play again? ") = 'n');
      END LOOP PlayAgain;
   END Play;
```

The askYesNo function asks a yes or no question. It keeps asking the question until the player enters either a y or an n and then returns either a 'y' or an 'n'. This is similar to the function in MadLib.

```
   FUNCTION AskYesNo (Question : String) RETURN Character IS
      Response : Character;
   BEGIN
      LOOP
         Put(Question); Put( " (y/n): ");
         Get_Immediate(Response); Put(Response);
         EXIT WHEN (Response = 'y' OR Response = 'n');
      END LOOP;
      New_Line;
      RETURN Response;
   END;
```

The askNumber Function asks for a number within a range and keeps asking until the player enters a valid number. It receives a question, a high number, and a low number. It returns a number within the range specified. It is similar to the equivalent function in MadLib.

```
   FUNCTION AskNumber (Question : String; High : Integer;
         Low : Integer := 1)
     RETURN Integer IS
      Number : Integer;
   BEGIN
      LOOP
         begin
            Put( Question ); Put( "("); Put(Low, 1);
            Put(" - "); Put(High, 1); Put("): ");
            Get(Number); Skip_Line;
            EXIT WHEN (Number <= High AND Number >= Low);
         EXCEPTION
            WHEN OTHERS => Put_Line("Pleasse enter an integer 1-9 only");
               skip_line;
         END;
      END LOOP;
      RETURN Number;
   END;
```

If you take a look at this function's spec, you can see that the low number has a default value of 1. I take advantage of this fact when I call the function later in the program.

The next function, humanPiece, asks the player if he/she wants to go first, again with some attitude, and returns the human's piece based on that choice. As the great tradition of Tic-Tac-Toe dictates, the player who goes first uses X.

```
FUNCTION HumanPiece RETURN Character IS
   Go_First : Character;
BEGIN
  Go_First := AskYesNo ("Do you require the first move?");
  IF (Go_First = 'y') THEN
      New_Line;
      Put_Line( "Then take the first move. You will need it.");
      RETURN X;
  ELSE
      New_Line;
      Put_Line("Your bravery will be your undoing. . . I will go first.");
      RETURN O;
  END IF;
END;
```

The opponent Function swaps players. It gets a piece (either an 'X' or an 'O') and returns the opponent's piece (either an 'O' or an 'X').

```
   FUNCTION Opponent ( Piece : Character) RETURN Character IS
   BEGIN
      IF (Piece = X) THEN RETURN O;
      ELSE RETURN X;
      END IF;
   END;
```

Note that in Ada2012 we could also use the shortened form if expression:

```
 c: character;
    BEGIN   -- if expressions added in 2012
       C := (IF Piece = O THEN X ELSE O);
       RETURN C;
  END;
```

The displayBoard Function displays the board passed to it. Because each element in the board is either a space, an 'X', or an 'O', the function can display each one. I use a few other characters on my keyboard to draw a decent-looking Tic-Tac-Toe board.

```
   PROCEDURE DisplayBoard ( Board : Board_Type) IS
   BEGIN
      New_Line;
      Put("    "); Put(Board(1));
      Put(" | ");  Put(Board(2));
      Put(" | ");  Put(Board(3));
      New_Line; Put_Line( "    ---------");
      Put("    "); Put(Board(4));
      Put(" | ");  Put(Board(5));
      Put(" | ");  Put(Board(6));
      New_Line; Put_Line( "    ---------");
      Put("    "); Put(Board(7));
      Put(" | ");  Put(Board(8));
      Put(" | ");  Put(Board(9));
      New_Line; Put_Line( "    ---------");
   END;
```

Note I do not use a For loop due to the special characters interspersed in the output. Keep it simple!

The winner Function receives a board and returns the winner. There are four possible values returned by the function: either X or O if one of the players has won, if every square is filled and no one has won, it returns TIE and finally, if no one has won and there is at least one empty square, the function returns NO_ONE.

The first thing to decide is how to pick a winner. Thinking about the game, I notice that a winner in defined as 3 of the same pieces in a line, in the 3 columns, or the 3 rows, or the two diagonals on the board. So there are 8 ways to get 3 in a line.

To represent this, I define a constant, two-dimensional array of integers called WINNING_ROWS, which represents all eight ways to get three in a line and win the game. Each winning line is represented by a group of three numbers—three board positions that form a winning line. For example, the group (1, 2, 3) represents the top row—board positions 1, 2, and 3. The next group, (4, 5, 6), represents the middle row—board positions 4, 5, and 6, and so on. . . .

After I have set up the Winning_Rows array, I next check to see whether either player has won. I loop through all 8 possible ways a player can win to see whether either player has three in a row. The if statement checks to see whether the three squares in question all contain the same value and are not EMPTY. If so, it means that the row has either three Xs or Os in it, and one side has won. The function then returns the piece in the first position of this winning row.

If neither player has won, I check for a tie game. If there are no empty squares on the board, then the game is a tie. I use a For loop to search the board for an EMPTY square and if one is found, there is no winner yet and the function returns NO_ONE. Finally, if there are no empty squares the game must be a TIE.

```
   FUNCTION Winner (
         Board : Board_Type)
     RETURN Character IS
      TYPE WINNING_ROWS_T IS ARRAY (1 .. 8, 1 .. 3) OF Integer;
      WINNING_ROWS : CONSTANT WINNING_ROWS_T :=
          ((1, 2, 3), (4, 5, 6), (7, 8, 9),
           (1, 4, 7), (2, 5, 8), (3, 6, 9),
           (1, 5, 9), (3, 5, 7));
      TOTAL_ROWS   : CONSTANT Integer := 8;
   BEGIN
      -- if any line has three values that are the same (and not EMPTY),
      -- then we have a winner
      FOR  Row IN 1.. TOTAL_ROWS LOOP
         IF ( ( Board(WINNING_ROWS(Row,1)) /= EMPTY ) AND
              ( Board(WINNING_ROWS(Row,1)) =
                  Board(WINNING_ROWS(Row,2)) ) AND
              ( Board(WINNING_ROWS(Row,2)) =
                  Board(WINNING_ROWS(Row,3)) ) ) THEN
            RETURN Board(WINNING_ROWS(Row,1)); -- we have a winner
         END IF;
      END LOOP;
      -- since nobody has won, check for a tie (no empty squares left)
      -- or no winner yet
      FOR I IN Board'First .. Board'Last LOOP
         IF ( Board(I) = EMPTY ) THEN
            RETURN NO_ONE; -- there is an empty square, so no winner yet
         END IF;
      END LOOP;
      -- if we get here, there are no empty squares, so the game is a TIE
      RETURN TIE;
   END;
```

This is a tricky bit of strategy and required some thought to set it up. In working it out, you might make several false starts before getting it right. This is often the case when designing a game – you may need to try several alternative approaches till you find one that works well for you.

The isLegal Function receives a board and a move. It returns true if the move is a legal one on the board or false if the move is not legal. A legal move is a move to an empty square.

```
   FUNCTION IsLegal(Board : Board_Type; Move : Integer) RETURN Boolean IS
   BEGIN
      RETURN (Board(Move) = EMPTY);
   END;
```

The test `(Board(Move) = EMPTY)` produces the required Boolean result to be returned.

Since the function turns out to be only one line long, it is a good candidate for inlining, so I added that aspect to the spec even though it will make little difference in such a simple game and most compilers are pretty good at optimising the code themselves anyway.

The humanMove Function receives a board and the human's piece. It returns the number of the square where the player wants to move. The function asks the player for the number to which he wants to move until the response is a legal move, then the function returns the move.

```
   FUNCTION HumanMove (Board : Board_Type) RETURN Integer IS
      Move : Integer := AskNumber ("Where will you move? ", NUM_SQUARES);
   BEGIN
      WHILE (NOT IsLegal(Board, Move)) LOOP
         New_Line;
         Put_Line( "That square is already occupied, foolish human." );
         Move := AskNumber("Where will you move?", NUM_SQUARES);
      END LOOP;
      Put_Line( "Fine. . ." );
      RETURN Move;
   END;
```

The function maintains the bit of attitude shown by the game towards the player.

I now come to the trickiest part of the game, the computerMove Function. This function receives the board and the computer's piece. It returns the computer's move.

Now on to the guts of the function. Okay, how do I program a bit of AI so the computer puts up a decent fight? Well, after playing a few games on paper, I came up with a basic three-step strategy for choosing a move. Examine the board and decide:

1. If the computer can win on this move, make that move.

2. Otherwise, if the human can win on his next move, block him/her.
3. Otherwise, take the best remaining open square. The best square is the centre. The next best squares are the corners, and then the rest of the squares.

The next section of the function implements Step 1.

```
FUNCTION ComputerMove (Board :  Board_Type; Computer :  Character)
  RETURN Integer IS
   Move  : Integer   := 1;
   Found : Boolean   := False;
   Human : Character;
   BoardCopy : Board_Type := Board;
BEGIN
   -- if computer can win on next move, that's the move to make
   WHILE (NOT Found AND (Move <= NUM_SQUARES)) LOOP
      BEGIN
         IF (IsLegal(BoardCopy, Move)) THEN
            BEGIN
               BoardCopy(Move) := Computer;
               Found := (Winner(BoardCopy) = Computer);
               BoardCopy(Move) := EMPTY;
            END;
         END IF;
         IF ( NOT Found) THEN
            Move := Move + 1;
         END IF;
      END;
   END LOOP;
```

The algorithm loops through all of the possible moves, 1–9. For each move, I test to see whether the move is legal. If it is, I put the computer's piece in that square and check to see whether the move gives the computer a win. If it does, I set the variable Found, then I undo the move by making that square empty again. If the move didn't result in a win for the computer, I go on to the next empty square. However, if the move did give the computer a win, then the loop ends—and I've found the move (found is true) that I want the computer to make that move to win the game.

One thing to notice is that I fiddle with the board, changing squares, testing if the result is a winning line, then restoring them. Since arguments in Ada are treated as constants within the function using them, I make a local copy of the board to play with. If the parameter were declared as IN OUT, it would be treated as a variable, but any changes made would then affect the original board in the procedure Play. This is not a good idea as it may cause side effects and possible malfunctions in the code. So, even though it's not as efficient to make a copy, by working with a copy I keep the original vector that represents the board safe.

At the end of this first part, if a winning move was not found (found is false), I then do step 2 of my AI strategy, which is to check if the human has a winning move, which I must block.

```
   -- otherwise, if human can win on next move,  then block that move
   IF (NOT Found) THEN
      BEGIN
         Move := 1;
         Human := Opponent(Computer);
         WHILE ( NOT Found AND (Move <= NUM_SQUARES) ) LOOP
            BEGIN
               IF (IsLegal(BoardCopy, Move)) THEN
```

```
                  BEGIN
                     BoardCopy(Move) := Human;
                     Found := (Winner(BoardCopy) = Human);
                     BoardCopy(Move) := EMPTY;
                  END;
               END IF;
               IF ( NOT Found) THEN
                  Move := Move + 1;
               END IF;
            END;
         END LOOP;
      END;
   END IF;
```

Again, I loop through all of the possible moves, 1–9. For each move, I test to see whether the move is legal. If it is, I put the human's piece in the corresponding square and check to see whether the move gives the human a win. Then I undo the move by making that square empty again. If the move didn't result in a win for the human, I go on to the next empty square. However, if the move did give the human a win, then the loop ends—and I've found the move (found is true) that I want the computer to make (square number move) to block the human from winning on his next move.

Next, I check to see if I need to go on to Step 3 of my AI strategy. If I haven't found a move yet (found is false), then I look through the list of best moves, in order of desirability, centre square first, then corners, then the rest, and take the first legal one. At this point I have found a legal move I want the computer to make—whether that's a move that gives the computer a win, blocks a winning move for the human, or is simply the best empty square available. So, I have the computer announce the move and return the move.

```
   -- otherwise, moving to the best open square is the move to make
   IF (NOT Found) THEN
      BEGIN
         DECLARE
            TYPE BEST_MOVES_T IS ARRAY (Integer RANGE <>) OF Integer;
            Best_Moves : CONSTANT Best_Moves_T :=
                     (5, 1, 3, 7, 9, 2, 4, 6, 8);
            I : Integer := Best_Moves'First;
            -- pick best open square
         BEGIN
            WHILE ((NOT Found) AND (I <= Best_Moves'Last)) LOOP
               Move := BEST_MOVES(I);
               IF (IsLegal(BoardCopy, Move)) THEN
                  Found := True;
               END IF;
               I := I+1;
            END LOOP;
         END;
      END;
   END IF;
   -- checked out all possible moves so announce computer's move
   Put( "I shall take square number ");
   Put( Move, 1 );
   New_Line;
   RETURN Move;
END;
```

Real World: The Tic-Tac-Toe game considers only the next possible move. Programs that play serious games of strategy, such as chess, look far deeper into the consequences of individual moves and consider many levels of moves and countermoves. In fact, good computer chess programs can consider literally millions of board positions before making a move.

The announceWinner procedure receives the winner of the game, the computer's piece, and the human's piece and announces the winner or declares a tie. It maintains its bit of attitude!

```
PROCEDURE AnnounceWinner (Winner, Computer, Human : Character) IS
BEGIN
  IF (Winner = Computer) THEN
    BEGIN
      Put_Line ( "I have won!");
      Put_Line ( "As I predicted, human, I win again – proof that");
      Put_Line ( "computers are superior to humans in all regards.");
      END;
  ELSIF (Winner = Human) THEN
    BEGIN
      Put_Line ( "You have won!");
      Put_Line ( "No, no! It can't be! Somehow you tricked me, human.");
      Put_Line ( "But never again! I, the computer, so swear it!");
    END;
  ELSE
    BEGIN
      Put_Line ( "It's a tie.");
      Put_Line ( "You were lucky, human, and managed to tie with me.");
      Put_Line ( "Celebrate . . this is the best you will achieve!");
    END;
  END IF;
END;
```

and that's it.

This chapter has looked at how you write subprograms and given several examples. It then went into issues of software design, using pseudocode and following that by planning a set of subprograms to implement the functionality of the program. In a later chapter we will study object oriented deign and programming, which will extend these ideas significantly.

For now, let's look at solving the problem raised in the player's database, that of saving the player data in a disk file.

9. Using disk files

All the programs so far have been "interactive", that is, the data was entered through keyboard and the results displayed on screen.

In many situations this is not suitable and we would prefer to read data from a file on the hard drive or memory stick that was created earlier, and have the results written to the file, to be printed or processed more a bit later. This will reduce the typing by allowing us to edit and check the data once, and then run the program multiple times. This is particularly valuable with large data sets. Using files also preserves the output as the screen is volatile, but files are persistent. Files are vital for games, where we wish to store the player's progress, all the players' scores and other details. In many games, too, we will read data from files in order to start the game.

When running interactively, you should always ensure what the user sees is fully self-explanatory by writing suitable messages, called prompts, to the display to guide the user.

For programs using files, when you read values from the files, information should be "echoed" to output so the user knows the program is running correctly.

Never leave the user staring at a blank screen!

Now let's try a couple of small programs to write and then read a file.

9.1 Create a text file and write to it

Reading and writing disk files in Ada is very simple, as the same Text_IO Get and Put procedures are used, with a file variable added. The procedure is to open the file for reading, writing or both and then proceed to read from it or write to it as desired. Here is a small example:

```
  --  Create a file and write some text to it.

with text_io; use text_io;

PROCEDURE WriteFile IS
   FileVar : File_type;   -- File_type is defined in Text_io
BEGIN
   Create( FileVar, Out_file, "MYTEXT.TXT" );
   Put_line( FileVar, "MYTEXT -- A sample text file." );
   Put_line( FileVar, " This program demonstrates how" );
   Put_line( FileVar, " to create and write to a disk" );
   Put_line( FileVar, " text file." );
   Close( FileVar );
   Put_line( "This program has created a text file called MYTEXT.TXT");
   Put_line( "Open it with a text editor to read it");
END;
```

Build and run the program and then look in the file MYTEXT.TXT that it created.

What happens if we run the program a second time? The file is reopened and overwritten! If you don't want to overwrite the file, add the package Ada.Directories and this code:

```
   IF Exists( "MYTEXT.TXT" ) THEN Put_Line("File already exists");
      <Or any other code>
   END IF;
```

9.2 Reading a text file

To read the file, open it in input mode and use Get to read it:

```
  -- Text file input.
  -- Read and display a text file line by line.
with text_io; use text_io;

PROCEDURE ReadFile IS
   FileVar  : File_type;   -- is text!
   OneLine  : String(1..80);
   Last     : integer;
BEGIN
   Open( FileVar, In_file, "MYTEXT.TXT" );
   WHILE NOT End_of_file( FileVar ) LOOP
      Get_line( FileVar, OneLine, Last );
      Put_line( OneLine( 1 .. Last ) );
   END LOOP;
   Close( FileVar );
END;
```

Run the program in the examples. You will get the text that was written by program WriteFile displayed on the screen. This shows a major use of file IO – it allows programs to communicate with each other via files, along with preserving data we do not want lost when a program ends. Note that we also use the optional parameter Last in Get and Put. Last contains the number of characters actually read in by Get and that is all we should write to the display. As the program reads the file, the file mode is set to In_file and the program reads and displays the file until the end of file is detected. Always close the file when you are finished with it.

Now delete file MYTEXT.TXT and run ReadFile again. What happens? As expected, you get an exception error displayed, that says MYTEXT.TXT: No such file or directory.

9.3 Formatting text files

Text files in Ada are considered to be just a stream of characters. Embedded characters mark the end of line and end of file, using the standard character coding. Users, however, like to see neat structure to files and to display outputs, so Text_IO also provides several procedures and functions for formatting (structuring) the output and managing the input:

SET_COL - go to nominated column in output file
NEW_LINE - go to next line of output
SET_LINE - go to nominated line in output file
NEW_PAGE - go to next page of output
SKIP_LINE - go to start of next line of input
SKIP_PAGE - go to start of next page of input
PAGE - what page number are we up to in the file?
LINE - what line number are we up to on the page?
COL - what character position are we up to on the line?
SET_LINE_LENGTH for lines
SET_PAGE_LENGTH for pages
Example:

```
SET_LINE (2);
SET_COL (30); PUT ("Student Results Report");
NEW_LINE (2);
SET_COL ( 5); PUT ("Student name");
SET_COL (35); PUT ("Assignments");
SET_COL (50); PUT ("Exams");
SET_COL (65); PUT ("Average");
```

9.4 File positioning

Often a file needs to be read thought twice, or, having been written, read back. With text files, as they have no structure, the only known positions are the beginning and the end of the file. A file can be set to the start of the file to re-read or re-write it, or the end of the file to append to the file.

```
RESET(FileVar, <mode>);  go back to the start of the file and (optionally) change mode.
        The file must be open already. Modes are In_File, Out_File, Append_File

E.g  RESET  (FileVar, Append_File);  set to end of file so text can be added to the file.
```

Files may be opened or created in any of the three above modes.

9.5 Direct access

Many applications need to be able to reposition quickly to any record (block of data) in the file. In this case the file must be created and written with fixed size records. The file then has a current position index through which you can access any record in the file to read or write to it.

Direct file access procedures

```
Open(FileVar, MODE, FileName);  -- modes are In_File, Inout_File, Out_File
Create(FileVar, MODE, FileName);
Close (FileVar);
Delete(FileVar);
Reset (FileVar, Mode );
function Mode(FileVar) return File_Mode;
function Name(FileVar) return String;
function Is_Open(FileVar) return Boolean;
Read(FileVar,  Item, <Index>);   -- Index is optional, read from next record if not given
Write(FileVar, Item, <index>);
Set_Index(FileVar, Index);
function Index(FileVar) return Positive_Count;  -- returns current index
function Size (FileVar) return Count;  -- reads file size as number of records
function End_Of_File(FileVar) return Boolean;
```

9.6 Sequential files

A sequential file, unlike a text file, is regarded as a sequence of values, which may be text, numbers, or structured data like records. This makes it useful to store records in sequence. Suppose, for example, we want to save the data from the players' database in a file before the program exits, and reload the saved data when we next run the program. This is very easy to accomplish with a sequential file.

9.7 Player database on file

Take a look at the player database program in Chapter 6. We can add a few lines to the program to store the player data in a file. The new main menu will have two new procedure calls, **`ReadMembers`** on startup, and **`WriteMembers`** before the program exits. We then add the two procedures to the package of subroutines called by the main program. We can also add an option, Save, to save intermediate work (which would be a useful feature to provide to the users).

```
BEGIN
   Put_line( "Player's Database" );
   ReadMembers;
   LOOP
      New_line;
      Put( "Number of players = " ); Put( Membership, 1 ); New_line;
      Put( "A.dd, C.harges, L.ist, P.ayments, S.ave to file, Q.uit ? " );
      Get( choice ); skip_line;
      CASE choice IS
        when 'A' | 'a' => AddRecords;
        when 'C' | 'c' => AddToCharges;
        when 'P' | 'p' => MakePayment;
        when 'L' | 'l' => ListRecords;
        when 'S' | 's' => WriteMembers;
        when 'Q' | 'q' => Exit;
        when others    => Put_line( "Invalid choice. Please try again." );
      END  case;
   END LOOP;
   WriteMembers;
END;
```

The procedure Readmembers will open the file of members records and read them into the array for processing. If a file does not exist, it creates a new one.

```
PROCEDURE ReadMembers Is
BEGIN
   Player_io.Open( MemberFile, Player_io.In_file, FileName );
   Membership := 0;
   WHILE NOT Player_io.End_Of_File( MemberFile ) LOOP
      membership := Membership + 1;
     Player_io.read( MemberFile, Members(Membership) );
   END LOOP;
   Player_io.Close( MemberFile );
   Put(Membership, 2); Put_line(" members records read in");
EXCEPTION
   When Player_io.NAME_ERROR =>  -- File does not exist
     Put_line( "New membership file." );
     Membership := 0;
END;
```

We have also put all the support subprograms in a separate package to keep the size of the main program small and to demonstrate this feature of Ada. In many large scale applications, there will be a large number of separate packages. Here there is one package and three files:

`PlayersWithFiles.adb` which is the main program,
`Player_Pack.ads` that contains the specifications of all the subprograms, and
`Player_Pack.adb` that contains their bodies (ie the code) of all the subprograms.

In the Player_Pack package, you will see that we first define the record type for the players, then have to create a specific version (an instance) of the Sequential_IO package, which we have named Player_IO, to handle those particular types of records for the program:

```
   package Player_io is new sequential_io(element_type => Player);
```

In a sequential access file, all the records must essentially be the same, although variant records may be used. A program can have multiple sequential files open, all handling different types, each with the own name, element type and IO package.

9.8 Direct access file example

The following sample program writes a file, then allows you to display and change any record in it. The file consists of a set of records, each containing a numerical student number followed by the student's name and gender. A file is created, containing 6 names. The list of students in this case is not kept in an array in the program, but is kept in a direct access file on disk. Note that the records are stored in binary computer code, so, except for the text components of the records, they do not make sense if looked at with a text editor. The records are listed on the display, two of the records are modified and the records are displayed again.

```
with text_io; use text_io;
WITH Ada.Direct_IO;
WITH Ada.Integer_Text_IO; USE Ada.Integer_Text_IO;

PROCEDURE FileDirectIO IS

Type GenType is (Male, Female, Unspecified);
Type StudentRec is
      RECORD
         Number    : Integer;
         Name      : String(1..30) := (others => ' ');  -- initialised
         Gender    : Gentype;
      END RECORD;

Package MyRecordIO is NEW Ada.Direct_IO(StudentRec); USE MyRecordIO; --
must make a specific instance of the package to handle StudentRec records.

   AStudent : StudentRec;
   RecordNumber : positive;
   BlankStr : constant String(1..30) := (OTHERS => ' ');
   FileVar : MyRecordIO.File_type;   -- File_type is defined as Direct_io
BEGIN
   Create( FileVar, InOut_File, "MyRecords.DAT" );
   AStudent.Number := 12323; AStudent.Name(1..9) := "Joe Boggs";
   AStudent.Gender := Male;
   Write( FileVar, AStudent); -- write first record
   AStudent.Name := BlankStr;
   AStudent.Number := 65783; AStudent.Name(1..10) := "Mary Helen";
   AStudent.Gender := Female;
   Write( FileVar, AStudent); --write second record, etc
   AStudent.Name := BlankStr;
   AStudent.Number := 65785; AStudent.Name(1..14) := "Fred Alexander";
   AStudent.Gender := male;
   Write( FileVar, AStudent);
   AStudent.Name := BlankStr;
   AStudent.Number := 32498; AStudent.Name(1..13) := "Susan Fellows";
   AStudent.Gender := Female;
   Write( FileVar, AStudent);
   AStudent.Name := BlankStr;
   AStudent.Number := 27456; AStudent.Name(1..12) := "Tracy Morgan";
   AStudent.Gender := Unspecified;
   Write( FileVar, AStudent);
   AStudent.Name := BlankStr;
   AStudent.Number := 49352; AStudent.Name(1..11) := "Casey Jones";
   AStudent.Gender := Male;
   Write( FileVar, AStudent);
   AStudent.Name := BlankStr;

   Put_line( " This program demonstrates how" );
   Put_line( " to create and write to a disk" );
   Put_line( " Direct Access file." );
   Put_line( "This program has created a text file called MYRecords.DAT");
   Put_line( "Now I will open it, read all the records and display them");
   Reset(FileVar);  -- go back to record 1
   recordNumber := 1;
   WHILE NOT End_Of_File(FileVar) LOOP
      Read(FileVar, AStudent);
      Put(recordNumber,3);Put(": ");Put(AStudent.Number); Put(" ");
      Put(AStudent.Name); Put(" ");Put(GenType'Image(Astudent.Gender));
      RecordNumber := RecordNumber + 1;
      new_line;
   end loop;
   Put_line( "Now I will change records 2 and 5 and re-list the records");
   Read(Filevar, Astudent, 2); -- read record 2
   AStudent.Name := BlankStr;
   AStudent.Name(1..12) := "Mary Marlane";  -- correct Student's Name
   Write(FileVar, AStudent, 2);  -- rewrite record 2
   Put_Line("Changed name in record 2")r
   Read(Filevar, Astudent, 5); -- read record 5
   AStudent.Gender := Female;    -- correct entry for student's gender
   Write(FileVar, AStudent, 5);  -- rewrite record 5
   Put_Line("Changed gender in record 5");
   Reset(FileVar);
   Put_Line("List all the records again");
   recordNumber := 1;
   WHILE NOT End_Of_File(FileVar) LOOP
      Read(FileVar, AStudent);
      Put(Recordnumber,3);Put(": ");Put(AStudent.Number); Put(" ");
      Put(AStudent.Name); Put(" ");Put(GenType'Image(Astudent.Gender));
      RecordNumber := RecordNumber + 1;
      new_line;
   end loop;
   Close(FileVar);
END;
```

There are some points to note in this example.

a) As with the sequential file example above, we have to create a specific version of the Direct_IO package, which we have named MyRecordIO, to handle the records for this program. In a direct access file, all the records must essentially be the same, although variant records may be used.
b) We can read or write any record in any order.
c) To display the Gender enumeration type, we use the attribute **`GenType'Image. The`** Image attribute generates the text associated with the enumeration type, so we do not need to create and import a compete instance of the enumeration_IO package.
d) The file can be any size, up to the system file size limit.

Let's now return to the Player's Database example. If the player data is saved in a Direct Access disk file, there would be several advantages. We could then list or change any member's record at will and update it on disk immediately. A direct access file will also extend automatically to accommodate any number of members. We could do away with the array of player records, and use the disk file to accommodate any number of players. I leave it to you, the interested reader, to modify the player database code to use direct file access.

10. Object oriented programming

Often in games (and of course in programming pretty much all applications) we encounter objects that have properties and activities we want to link to the object. An example might be a vehicle with a number of wheels, a load carrying capacity and activities like starting, driving, etc. We may also want to build new instances of the object. Object-oriented programming (OOP) is a different way of thinking about programming, in terms of such objects. It's a modern methodology that's used in the creation of the vast majority of games (and other commercial software, too). In OOP, you define different types of objects with relationships to each other that allow the objects to interact. You've already worked with objects from types defined in libraries, such as the IO libraries and maths library, but one of the key characteristics of OOP is the ability to make your own types from which you can create objects. In this chapter, you'll see how to define your own types and create objects from them.

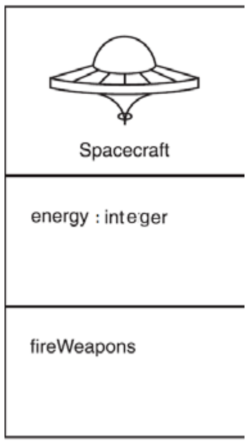
10.1 Understanding Objects

Most of the things you want to represent in games—such as, say, an alien spacecraft—are objects. They're cohesive things that combine qualities (such as an energy level) and abilities (for example, firing weapons). Often it makes no sense to talk about the individual qualities and abilities in isolation from each other.

Fortunately, most modern programming languages let you work with software objects (often just called objects) that combine data and subprograms. A data element of an object is called a data member, while a subprogram of an object is called a member subprogram, or a method. As a concrete example, think about that alien spacecraft. An alien spacecraft object might be of a new type called Spacecraft, defined by a game programmer, and might have a data member for its energy level and a member function to fire its weapons. In practice, an object's energy level might be stored in its data member energy as an integer, and its ability to fire its weapons might be defined in a member procedure called fireWeapons.

Every object of the same type has the same basic structure, so each object will have the same set of data members and member subprograms. However, as an individual, each object will have its own values for its data members. If you had a squadron of five alien spacecrafts, each would have its own energy level. One might have an energy level of 75, while another might have a level of only 10, and so on. Even if two crafts have the same energy level, each would belong to a unique spacecraft. Each craft could also fire its own weapons with a call to its member function, fireWeapons.

Objects are often shown on diagrams like this one, which illustrates the concept of an alien spacecraft.



Spacecraft object

The cool thing about objects is that you don't need to know the implementation details to use them—just as you don't need to know how to build a car in order to drive one. You only have to know the object's data members and member functions—just as you only need to know where a car's steering wheel, gas pedal, and brake pedal are located.

You can store objects in variables, just like with built-in types. Therefore, you could store an alien spacecraft object in a variable of the Spacecraft type.

Once you create a spacecraft type, you can declare objects of that type. For example:

```
      Ship1, Ship2 : Spacecraft;
      Fleet is array (1..5) of Spacescraft;
```

You can access data members and member functions using the member selection operator (.), by placing the operator after the variable name of the object. So if you want your alien spacecraft, ship, to fire its weapons only if its energy level is greater than 10, you could write:

```
      -- ship1 is an object of Spacecraft type
      if (ship1.energy > 10) then
         ship1.fireWeapons;
      endif;
```

This kind of usage is familiar – it is the way we have used packages from the library, with the data and the subprograms defined in the package epecification and the implementation in the body. So, in Ada, objects are built using packages. Specifically, we will:

- Create new types in packages
- Declare package data members and member functions
- Instantiate objects from the packages
- Set member access levels

Whether you're talking about alien spacecraft, poison arrows, or angry mutant chickens, games are full of objects. Fortunately, Ada lets you represent game entities as software objects, complete with member functions and data members. These objects work just like the others you've already seen, such as string and vector objects. But to use a new kind of object (say, an angry mutant chicken object), you must first define a type for it, as the Spaceship example will demonstrate. This is done in the package specification, where the data members are declared in a record. The name of the record then becomes the name of the object:

```
PACKAGE Spaceships IS  -- package definition -  defines a new type, Critter

   TYPE Spaceship IS
      RECORD
         Energy : Integer := 50; -- data member with initial value
      END RECORD;

   PROCEDURE FireWeapons(Ship : in out Spaceship); -- member subprogram

END Spaceships;
```

Note in this example the data member item Energy is given a default value of 50.

The body of the object takes the form of a package body. It contains just one procedure, FireWeapons(Ship : in out Spaceship), where the argument Ship identifies which ship is firing.

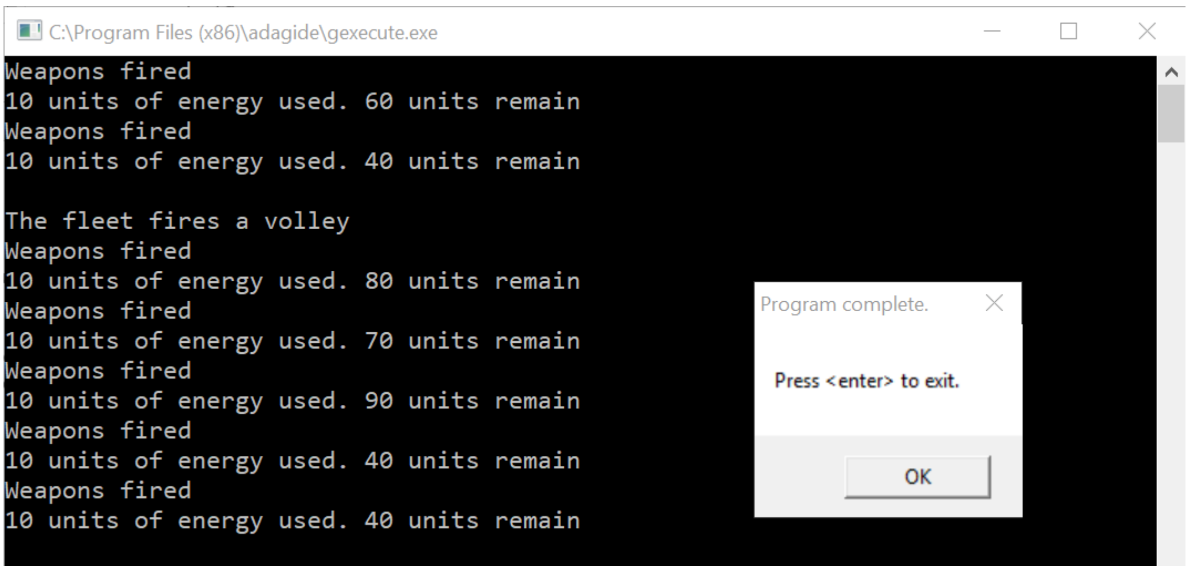
```
WITH Text_Io; USE  Text_IO;
With Ada.Integer_Text_IO; Use Ada.Integer_Text_IO;

PACKAGE BODY Spaceships IS  --  Spaceship object implementation
```

```
   PROCEDURE FireWeapons(Ship : in out Spaceship) IS
   BEGIN
      Put_Line("Weapons fired");
      Put("10 units of energy used. ");
      Ship.Energy := Ship.Energy - 10;
      Put(Ship.Energy, 2);
      Put_line(" units remain");
   END;

END Spaceships;
```

Having built the Spaceship object, we can write a main program to create some spaceship object instances and have them fire their weapons. In this example, we make two individual instances plus a fleet of 5 ships as an array of 5 Spaceship objects. We also initialise the energy levels of some of them, leaving the others with their default energy levels. In the example below, Ship1 : Spaceship := (Energy => 70); sets the energy level of the instance Ship1 to 70. The example also initialises two of the ships in the fleet using the usual item by tiem method of initializing array elements covered in chapter 4.

```
WITH Ada.Text_IO; USE Ada.Text_IO;
WITH Spaceships; USE Spaceships;

PROCEDURE SpaceOdyssey IS
   Ship1 : Spaceship := (Energy => 70); -- declared and initialised
   Ship2 : Spaceship;                   -- declared with default value
   Fleet : ARRAY (1 .. 5) OF Spaceship :=
                  ((Energy => 90), (Energy => 80), OTHERS => <> );
   -- declare a fleet of 5 ships, initialise the first 2, leave the rest
   -- with their default values, indicated by the notation <>
BEGIN
   Fireweapons(Ship1);
   IF Ship2.Energy > 10 THEN
      Fireweapons(Ship2);
   END IF;
   New_Line;
   -- Give third ship in fleet more energy
   Fleet(3).Energy := 100;
   Put_Line("The fleet fires a volley");
   FOR Ships IN Fleet'RANGE LOOP
      Fireweapons(Fleet(Ships));
   END LOOP;

END SpaceOdyssey;
```

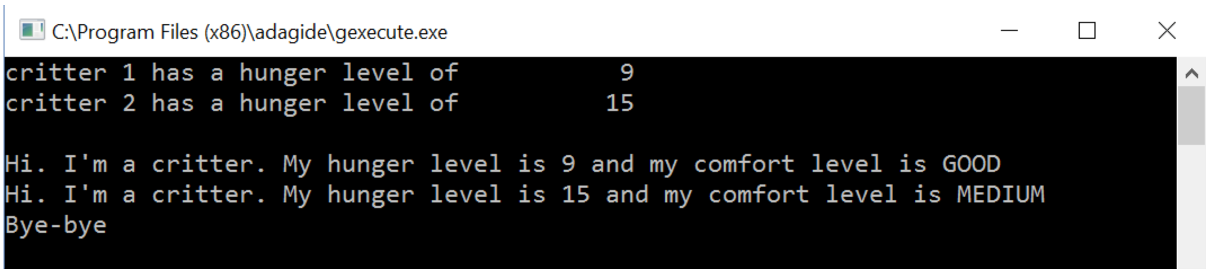
Here is the output:



```
Weapons fired
10 units of energy used. 60 units remain
Weapons fired
10 units of energy used. 40 units remain

The fleet fires a volley
Weapons fired
10 units of energy used. 80 units remain
Weapons fired
10 units of energy used. 70 units remain
Weapons fired
10 units of energy used. 90 units remain
Weapons fired
10 units of energy used. 40 units remain
Weapons fired
10 units of energy used. 40 units remain
```

10.2 Introducing the Simple Critter Program

We will now build a series of games based on the the idea of the virtual pet, or critter, and use these to illustrate more concepts of objects. We will begin with a Simple Critter Program that defines a brand-new type called Critter for creating our virtual pet objects. The program uses this new type to create two Critter objects. Then, it gives each critter a hunger and a comfort level. Finally, each critter offers a greeting and announces its hunger and comfort levels to the world. Here is the result of running the program:



```
critter 1 has a hunger level of          9
critter 2 has a hunger level of         15

Hi. I'm a critter. My hunger level is 9 and my comfort level is GOOD
Hi. I'm a critter. My hunger level is 15 and my comfort level is MEDIUM
Bye-bye
```

10.3 Defining an object

As we saw with the Spaceship example, to create a new object, you define a package — code that groups data members the and member subprograms (procedures and functions). The data members are grouped as a record type, the name of which is the name of the object. The subprograms (also called methods in Object terminology) are gouped in the package with the record holding the data members. Using the package, you create individual objects, called instances, that have their own copies of the data member record and access to all of the member subprograms. The package is like a blueprint. Just as a blueprint defines the structure of a building, a package defines the structure of an object. And just as a foreman can create many houses from the same blueprint, a game programmer can create many objects from the same package. Some real code will help solidify this theory. The package, as usual, consists of two files, the specification and the body. Let’s start with an object definition in the Simple Critter program with the specification for the package. My convention is to use an uppercase letter and the plural form for package names, and the package spec defines the object.

10.4 Declaring Data Members

In an object specification, you can declare data members to represent object qualities. I give the critters two qualities, hunger and comfort. The data members are normally declared as a record to group them together. I see hunger as a range that could be represented by an integer, so I declare an integer data member, Hunger. This means that every Critter object will have its own hunger level, represented by its own data member named Hunger. I also declare a comfort level, which I see as poor, medium, good or great, for which I create and then use an enumeration type to define the data member, Comfort. Data members can be given default values, and here I give Comfort the default level of Medium. Many gamers follow a convention of prefixing object data members names with a prefix like m_ so that data members are instantly recognizable. Others rely on the object name, which is what I do here.

10.5 Declaring Member Abilities

In an object definition, you can also declare member subprograms to represent object abilities. I give Critters just one—the ability to greet the world and announce its hunger and comfort levels—by declaring the member procedure Greet. This means that every Critter object will have the ability to say hi and announce its own hunger level through its member procedure, Greet. By convention, member subprogram names begin with an uppercase letter. At this point, I've only declared the member function Greet. Don't worry, though, I'll define it outside of the class.

```
PACKAGE Critters IS  -- package definition -  defines a new type, Critter

   Type ComfortT is (poor, medium, good, great);

   TYPE Critter IS
      RECORD
      Hunger : Integer; -- data member
      Comfort : ComfortT := medium; -- data member with default value
      END RECORD;

   PROCEDURE Greet(Acritter : critter);  -- member function definition

END Critters;
```

10.6 Writing the package body

The next step is to write the body of the package. I'll do this in the same way as before, so it looks other package bodies you've seen, except for one thing — I add an argument of type Critter to the procedure Greet, which will cause the correct critter to be selected when it is called.

In the member function, I send Hunger and Comfort to the display with the usual IO calls of Put. This means that Greet displays the values of Hunger and Comfort for the specific object through which the function is called. You can access the data members of an object by qualifying it with the name of the object instance and you can call the member functions of an object in any member function simply by using the desired object instance as an argument to the call.

The body of critters looks like this:

```
WITH Text_Io; WITH Ada.Integer_Text_IO;
USE  Text_IO; USE  Ada.Integer_Text_IO;
PACKAGE BODY Critters IS  --  Critter implementation
   PROCEDURE Greet(Acritter : critter) IS
   BEGIN
      Put( "Hi. I'm a critter. My hunger level is ");
      Put( Acritter.Hunger, 1 );
      Put (" and my comfort level is ");
      put( ComfortT'image(Acritter.comfort));
      New_Line;
   END;

END Critters;
```

Note I use ComfortT'image to 'put' Comfort. This outputs the string representing the variable so I do not need to 'with' or make an instance of the enumeration_io package.

10.7 Using the package.

Now we can write some code to use the package. First we 'with' the package (as usual) and also 'with' the library packages we need. Then we make two instances of Critter, Crit1 and Crit2. As a result, we have two Critters, each with their own data. Note how I initialise Crit1 when I instantiate (make an instance) of it. This will override any defaults. When you instantiate objects, you often want to do some initialization—usually assigning values to data members. Luckily, this is easy to do and is a great convenience. For more complex objects, Ada allows fully-general constructor functions, so when a complex object is instantiated it can be fully initialised and default start-up code executed.

Since we have made the Critters package visible, it is simple to access the data for Crit1 and Crit2, just by writing, e.g. Crit2.Hunger := 15; This code assigns 15 to crit2's data member Hunger. Just like when you are accessing an available member function of an object, you can access an available data member of an object using the member selection operator. To prove that the assignment worked, I display hunger levels for both critters, which correctly shows 9 and 15, so, crit1 and crit2 are both instances of Critter, and yet each exists independently and each has its own identity. Also, each has its own Hunger and Comfort data members with their own values.

Next I call the Greet member procedure for Crit1 and Crit2. The call chooses the correct function according to its argument, so displays the correct values for the two Critters. Note that if I did not say USE Critters; in the first line, then I would have to qualify both the data member access and the procedure call by saying e.g. Critters.Crit2.Hunger for the data and Critters.Greet(Crit1), etc. If there any confusion about which object I want, I can always fully qualify it to identify the right one.

Here is the code for the main procedure myCritters that uses the Critters package:

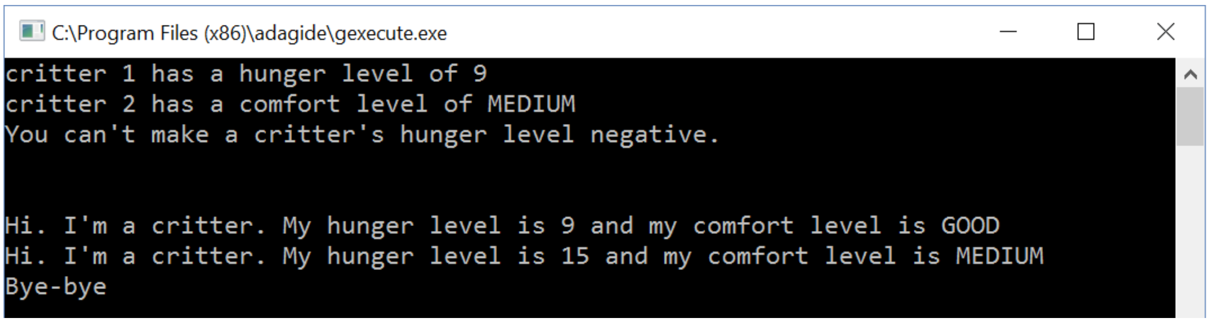
```
WITH Critters; USE Critters;
WITH Text_IO;  USE Text_Io;
with Ada.Integer_Text_IO; use ada.Integer_Text_IO;
```

```
PROCEDURE MyCritters IS
   Crit1 : Critter := (Hunger => 9, Comfort => Good ); -- initialised
   Crit2 : Critter;                                 -- uninitialised instance
BEGIN
   -- can access Critter's visible data with '.' notation
   Crit2.Hunger := 15;
   put("critter 1 has a hunger level of "); put(crit1.Hunger); new_line;
   put("critter 2 has a hunger level of "); put(crit2.Hunger); new_line(2);
   -- use Critter's built-in procedure
   Greet(Crit1);
   Greet(Crit2);
   Text_Io.Put("Bye-bye");
END MyCritters;
```

10.8 Setting Member Access Levels - the Private Critter Program

Like subprograms, you should treat objects as encapsulated entities. This means that, in general, you should avoid directly altering or accessing an object's data members. Instead, you should call an object's member subprograms, allowing the object to maintain its own data members and ensure their integrity. Fortunately, you can enforce data member restrictions when you specify and object package by setting member access levels.

The Private Critter program demonstrates object member access levels by specifying access levels for critters that restricts direct access to an object's data members for its hunger and comfort levels. The package provides two member functions for each data member—one that allows access to the data member and one that allows changes to the data member. The program creates a new critter and indirectly accesses and changes the critter's hunger and comfort levels through these member functions. However, when the program attempts to change the critter's hunger level to an illegal value, the member function that allows the changes catches the illegal value and doesn't make the change. Finally, the program uses the hunger-level-setting member function with a legal value, which works like a charm. Here are the results of running the program:


C:\Program Files (x86)\adagide\gexecute.exe

```
critter 1 has a hunger level of 9
critter 2 has a comfort level of MEDIUM
You can't make a critter's hunger level negative.


Hi. I'm a critter. My hunger level is 9 and my comfort level is GOOD
Hi. I'm a critter. My hunger level is 15 and my comfort level is MEDIUM
Bye-bye
```

Here is the spec:

```
PACKAGE PrivCritters IS  -- package definition -  defines a new type,
Critter
   TYPE ComfortT IS           (Poor, Medium, Good, Great);
   TYPE Critter IS PRIVATE;

   PROCEDURE Greet (Acritter : Critter);-- member function definition
   PROCEDURE PutHunger (ACrit : IN OUT Critter; H : Integer);
   PROCEDURE PutComfort (ACrit : IN OUT Critter; C : ComfortT);
   FUNCTION GetHunger (ACrit : Critter) RETURN Integer;
   FUNCTION GetComfort ( ACrit : Critter) RETURN ComfortT;

PRIVATE  -- begin private section
   TYPE Critter IS
      RECORD
         Hunger  : Integer;            -- data member
         Comfort : ComfortT := Medium;
      END RECORD;

END PrivCritters;
```

10.9 Specifying Public and Private Access Levels

Every class data member and member function has an access level, which determines from where in your program you can access it. So far, I've always left the members with their default public access levels. This means that any data member or member function that follows (until another access level specifier) will be public. This means that any part of the program can access them. Because I declare all of the member functions in this section, it means that any part of my code can call any member function through a Critter object. In this new object package, I leave the Comfort Type ComfortT public, then I specify Type Critter as private. This is followed by the specs for the public member functions.

Next, I specify a private section with the following line:

```
PRIVATE  -- begin private section
```

By using private:, I'm saying that any data member or member function that follows (until another access level specifier) will be private. This means that only code in the Critter package body can directly access it. Since I declare Hunger and Comfort in this section, it means that only the code in PrivCritter can directly access an object's Hunger or Comfort data member. Therefore, I can't directly access an object's Hunger data member through the object in the main procedure as I've done in previous programs. So the following line in the main procedure would be an illegal statement:

```
Crit1.Hunger := 15;
```

Because Hunger is private, I can't access it from code that is not part of the Critter package. Again, only code that's part of Critter can directly access the data member.

I've only shown you how to make data members private, but you can make member functions private, too. There are also other access levels, such as limited private, of which more later. Finally, member access is public by default. Until you specify an access modifier, any class members you declare will be public. By convention, the definition of the private data is placed at the end.

10.10 Coding the member functions

Now let's look at the package body, which implements the member subprograms that allow indirect access to a data member. Because Hunger and comfort are private, I wrote functions, GetHunger and getComfort to return the values of the data members. I also wrote two procedures, putHunger and putComfort, to set the values in Hunger and Comfort. The putHunger procedure, in addition, checks that the hunger level being set is not negative and displays an error message if it is. Another way of dealing with an illegal value would be to raise an exception, to be dealt with in the client program.

Here is the code for the body:

```
PACKAGE BODY PrivCritters IS  --  Critter implementation

   PROCEDURE Greet(Acritter : critter) IS
   BEGIN
      Put( "Hi. I'm a critter. My hunger level is ");
      Put( Acritter.Hunger, 1 );
      Put (" and my comfort level is ");
      put( ComfortT'image(Acritter.comfort));
      New_Line;
   END;

   PROCEDURE PutHunger(ACrit : in out Critter; H : Integer) IS
   BEGIN
      if H < 0 then
         put_line("You can't make a critter's hunger level negative.");
      ELSE ACrit.Hunger := H;
      end if;
   END;

   PROCEDURE PutComfort(ACrit : in out critter; C : ComfortT) is
   BEGIN
      ACrit.Comfort := C;
   END;

   FUNCTION GetHunger(ACrit : Critter) RETURN Integer IS
   BEGIN
      RETURN Acrit.Hunger;
   END;

   Function getComfort(ACrit : critter) return comfortT is
   BEGIN
      RETURN Acrit.Comfort;
   END;

END PrivCritters;
```

At this point, you might be wondering why you'd go to the trouble of making a data member private only to grant full access to it through accessor functions. One answer is that you don't generally grant full access. For example, take a look at the member procedure I defined for setting an object's Hunger data member, putHunger, where I first check to make sure that the value passed to the member function is greater than zero. If it's not, it's an illegal value and I display a message, leaving the data member unchanged. If the value is greater than zero, then I make the change. This way, SetHunger protects the integrity of Hunger, ensuring that it can't be set to a negative number. Another reason is that allowing access to internal data would mean that, if the implementation of the package body changed, all the client procedures would have to be rewritten. By hiding the implementation (encapsulation), the clients and the package spec need not be changed if the package body were changed, so long as the body remained compliant with the spec. Why might the package body be changed, you might ask? This could be to correct errors, to implement more efficient algorithms or because some other package that this package body uses is changed. A further reason is encapsulation allows for greater object flexibility though extension, inheritance and polymorphism, which we will study later and which would not be possible without encapsulation.

For now, let’s look at the client procedure:

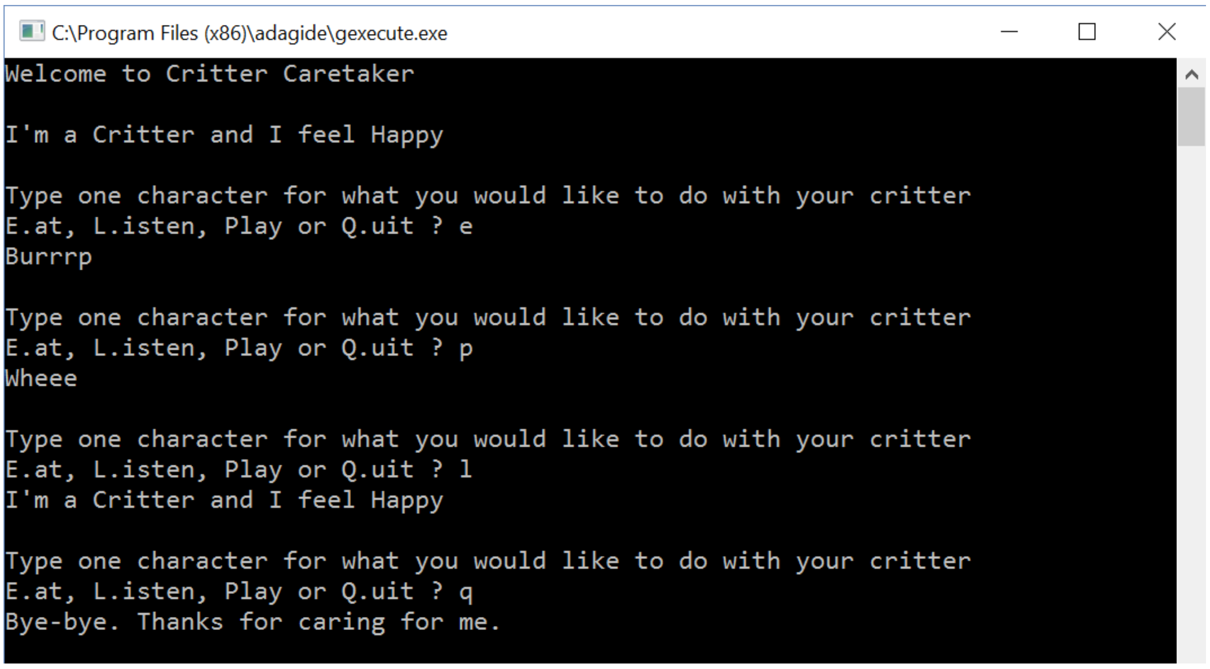
```
WITH PrivCritters; USE PrivCritters;
WITH Text_IO; USE Text_Io;
WITH Ada.Integer_Text_IO; USE Ada.Integer_Text_IO;

PROCEDURE MyPrivCritters IS
   Crit1 : Critter; -- can't initialise instances
   Crit2 : Critter; -- since the data is private
BEGIN
   -- can't access Critter's data directly
   PutHunger(Crit1, 9);
   PutComfort(Crit1, Good);
   PutHunger(Crit2, 15);
   Put("critter 1 has a hunger level of ");
   Put(getHunger(Crit1), 1);
   New_Line;
   Put("critter 2 has a comfort level of ");
   Put(ComfortT'Image(GetComfort(Crit2)));
   new_line;
   putHunger(Crit1, -1);
   New_Line(2);
   -- use Critter's built-in greet procedure
   Greet(Crit1);
   Greet(Crit2);
   Put("Bye-bye");
END MyPrivCritters;
```

I start by ‘with’ing the packages I will be using, including PrivCritters, of course. Next, I create two instances, Crit1 and Crit2, as before. I can’t use the simple initialisation I used in the previous version because the data is private, so I set the values of Hunger and Comfort in Crit1 and Hunger in Crit2, but leave the comfort level of Crit2 at its default level of medium. Then I display the hunger level of Crit1 and the Comfort level of Crit2. For the latter I use the ComfortT'Image built in attribute to save me implementing an instance of the enumeration_IO package. Then I try to set Crit1’s hunger level to -1 (an illegal value) which results is rejected with an error message. I have done this simply to show you how such a function might be written. In practice, in Ada, if a number had to be positive, we should declare it as positive; then any such attempt would be caught as an error by the compiler. Finally I call greet which displays the values of hunger and comfort for the two critters, as before.

10.11 The Critter Caretaker

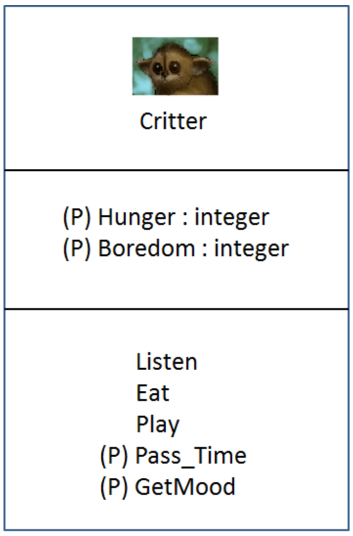
The Critter Caretaker game puts the player in charge of his own virtual pet. The player is completely responsible for keeping the critter happy, which is no small task. He can feed and play with the critter to keep it in a good mood. He can also listen to the critter to learn how the critter is feeling.

Here is the output from the game:



```
Welcome to Critter Caretaker

I'm a Critter and I feel Happy

Type one character for what you would like to do with your critter
E.at, L.isten, Play or Q.uit ? e
Burrrp

Type one character for what you would like to do with your critter
E.at, L.isten, Play or Q.uit ? p
Wheee

Type one character for what you would like to do with your critter
E.at, L.isten, Play or Q.uit ? l
I'm a Critter and I feel Happy

Type one character for what you would like to do with your critter
E.at, L.isten, Play or Q.uit ? q
Bye-bye. Thanks for caring for me.
```

10.12 Planning the Game



The core of the game is the critter itself. Therefore, I first plan my Critter package. Because I want the critter to have independent hunger and boredom levels, I know that the class will have private data members for those.

The critter should also have a mood, directly based on its hunger and boredom levels. My first thought was to have a private data member, but a critter's mood is really a calculated value based on its hunger and boredom. Instead, I decided to have a private member subprogram, getMood, that calculates a critter's mood on the fly, based on its current hunger and boredom levels.

Next, I think about public member subprograms. I want the critter to be able to tell the player how it's feeling. I also want the player to be able to feed and play with the critter to reduce its hunger and boredom levels. I need three public member subprograms to accomplish each of these tasks, Talk, Eat and Play

Finally, I want another member subprogram that simulates the passage of time, to make the critter a little more hungry and bored: PassTime. I see this member subprogram as private because it will only be called by other member subprograms, such as Talk, Eat, or Play.

Here is a diagram of the package, using a common style of notation.

At the top is the name (the picture is optional), in the middle box is the data, with private items marked (P) and the bottom box lists the subprograms, again with the private items marked (P).

10.12 Planning the Pseudocode

The rest of the program will be pretty simple. It'll basically be a game loop that asks the player whether he wants to listen to, feed, or play with the critter, or quit the game. Here's the pseudocode I came up with:

```
Create a critter
While the player doesn't want to quit the game
   Present a menu of choices to the player
   If the player wants to listen to the critter
      Make the critter talk
   If the player wants to feed the critter
      Make the critter eat
   If the player wants to play with the critter
      Make the critter play
```

The main procedure, CritterCaretaker, follows the above design closely and uses the same menu format used in the player's database example. I start as usual by 'with'ing the packages I need, then I instantiate a new Critter object. Because I don't supply values for Hunger or Boredom, the data members start out at 0, and the critter begins life happy and content. Next, I create a menu system. If the player enters Q, the program ends. If the player enters L, the program calls the object's Talk member procedure. If the player enters E, the program calls the object's Eat member procedure. If the player enters P, the program calls the object's Play member procedure. If the player enters anything else, he is told that the choice is invalid. Here it is:

```
WITH CritterPak; USE CritterPak;
WITH Text_IO; USE Text_Io;

PROCEDURE CritterCaretaker IS
   Crit : Critter;
   Choice : Character;
BEGIN
   Put_Line("Welcome to Critter Caretaker"); New_Line;
   Talk(Crit); -- Critter says Hi
   LOOP
      New_line;
      Put_line("Type one character for what you would like to do with your
critter");
      Put( "E.at, L.isten, Play or Q.uit ? " );
      Get( Choice );  Skip_Line;

      CASE choice IS
        when 'E' | 'e' => Eat(Crit);
        when 'L' | 'l' => Talk(Crit);
        when 'P' | 'p' => Play(Crit);
        when 'Q' | 'q' => Exit;
        when others    => Put_line( "Invalid choice. Please try again with
E, L, P or Q." );
      END  case;
   END LOOP;

   Put("Bye-bye. Thanks for caring for me.");
END CritterCaretaker;
```

Next I design the spec for the critter package, which I decided to call CritterPak. Again, it closely follows my description of the Critter's data and subprograms in the description of the object. Here is the code:

```
PACKAGE CritterPak IS  -- package definition -  defines a new type, Critter

   TYPE Critter IS PRIVATE;

   PROCEDURE Eat (Crit : IN OUT Critter; Food : Integer := 4);-- public
   PROCEDURE Talk (Crit : IN OUT Critter);     -- subprogram definitions
   PROCEDURE Play (Crit : IN OUT Critter; fun : integer := 4);

PRIVATE
   TYPE Critter IS
      RECORD            -- Private Critter Data
         Hunger  : Integer := 0;
         Boredom : Integer := 0;
      END RECORD;
   FUNCTION GetMood(Crit : Critter) return Integer;  -- Private subprograms
   procedure passTime(Crit : in out Critter; SomeTime : Integer := 1);

END CritterPak;
```

Now I will build the bodies of the subprograms defined in the spec. The package body starts as usual:

```
WITH Text_Io; USE  Text_IO;

PACKAGE BODY CritterPak IS  --  Critter implementation
```

The GetMood Member Function

The implementation in the package body opens with the GetMood function:

```
   FUNCTION GetMood(Crit : Critter) Return Integer IS
   BEGIN
      RETURN Crit.Hunger + Crit.Boredom;
   END;
```

The function returns a value that represents a critter's mood, as the sum of a critter's hunger and boredom levels. The critter's mood gets worse as the number increases. I made this member function private because it should only be invoked by another member subprogram in the package. I did not make the argument IN OUT since it doesn't change the data members, and the default parameter mode of IN treats the argument as constant within the function.

The PassTime Member Procedure

PassTime is a private member procedure that increases a critter's hunger and boredom levels. It's invoked at the end of each member subprogram where the critter does something (eats, plays, or talks) to simulate the passage of time. I made this member procedure private because it should only be invoked by another member subprogram in the package.

```
   PROCEDURE PassTime(Crit : in out Critter; SomeTime : Integer := 1) IS
   BEGIN
      Crit.Hunger := Crit.Hunger + SomeTime;
      Crit.Boredom := Crit.Boredom + SomeTime;
   END;
```

You can pass the member procedure the amount of time that has passed, otherwise, time gets the default argument value of 1, which I specify in the procedure specification.

The critter's mood gets worse each time the procedure is called, since the mood is the sum of the critter's hunger and boredom levels. I made this member procedure private because it should only be invoked by another member subprogram in the package. I made it argument Crit IN OUT since its data members are changed.

The Talk Member Procedure

The Talk member Procedure announces the critter's mood, which can be happy, okay, frustrated, or mad. Talk calls GetMood and, based on the return value, displays the appropriate message to indicate the critter's mood. Finally, Talk calls PassTime to simulate the passage of time.

```
PROCEDURE Talk(Crit : in out critter) IS
Mood : constant integer := GetMood(Crit);
BEGIN
   Put("I'm a Critter and I feel ");
   CASE Mood IS
      WHEN 0..5 => Put_Line("Happy");
      WHEN 6..10 => Put_Line("Okay");
      WHEN 11..15 => Put_Line("Frustrated");
      when others => Put_line("Angry");
   END CASE;
   PassTime(Crit);
END;
```

The Eat Member Procedure

Eat reduces a critter's hunger level by the amount passed to the parameter food. If no value is passed, food gets the default argument value of 4. The critter's hunger level is kept in check and is not allowed to go below zero. Finally, PassTime is called to simulate the passage of time.

```
PROCEDURE Eat(Crit : in out Critter; Food : Integer := 4) IS
BEGIN
   Put_Line("Burrrp");
   Crit.Hunger := Crit.Hunger - Food;
   IF Crit.Hunger < 0 THEN
      Crit.Hunger := 0;
   END IF;
   PassTime(Crit);
END;
```

The Play Member Procedure

Play reduces a critter's boredom level by the amount passed to the parameter fun. If no value is passed, fun gets the default argument value of 4. The critter's boredom level is kept in check and is not allowed to go below zero. Finally, PassTime is called to simulate the passage of time.

```
PROCEDURE Play(Crit : in out Critter; Fun : Integer := 4) IS
BEGIN
   Put_Line("Wheee");
   Crit.Boredom := Crit.Boredom - Fun;
   IF Crit.Boredom < 0 THEN
      Crit.Boredom := 0;
   END IF;
```

```
      PassTime(Crit);
   END;
```

Exercise: As an interesting variation, PassTime could read the time of day clock and make the increase in Boredom and Hunger depend on the time since it was last called. Then, if the Critter is neglected for a long time, they could increase by a large amount and if this exceeds some threshold, the Critter might die.

11. Extending objects

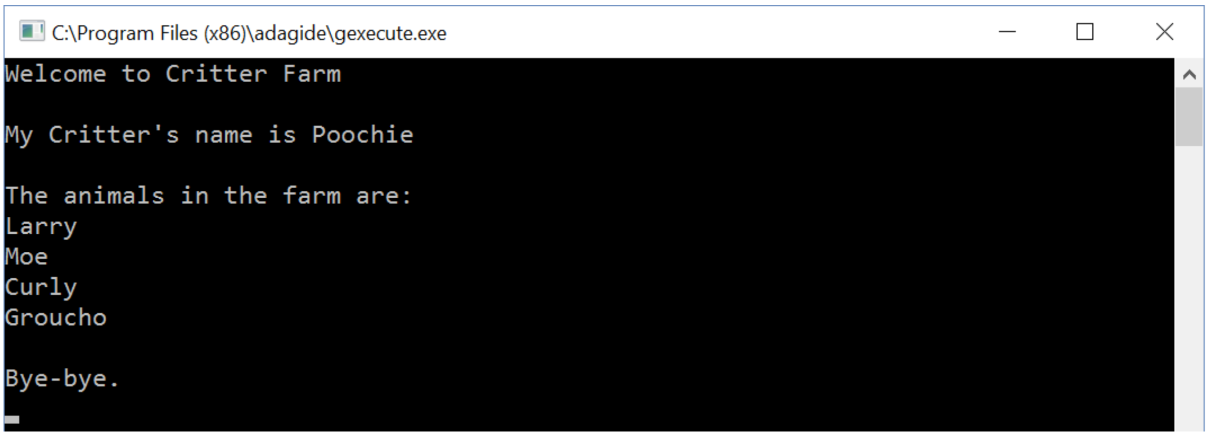
So far, we have only used Ada's most basic form of object. For serious object oriented programming, we need to allow objects to be extended and have great flexibility. Let's now look at a further example to show how objects can be extended using aggregation.

11.1 Using Aggregation

Game objects are often composed of other objects. For example, in a racing game, a drag racer could be seen as a single object composed of other individual objects, such as a body, four tires, and an engine. Other times, you might see an object as a collection of related objects. In a zookeeper simulation, you might see the zoo as a collection of an arbitrary number of animals. You can mimic these kinds of relationships among objects in OOP using aggregation—the combining of objects so that one is part of another. For example, you could write a Drag_Racer class that has an engine data member that's an Engine object. Or, you could write a Zoo class that has an Animals data member that is a collection of Animal objects. To illustrate these ideas, I will build a Critter Farm.

11.2 Introducing the Critter Farm Program

The Critter Farm program defines a new kind of critter with a name. First the program make a single critter, as we have seen before. Then, after the program announces the new critter's name, it creates a critter farm as a collection of critters. Finally, the program calls roll on the farm and each critter announces its name. Here's the result of running the program.



C:\Program Files (x86)\adagide\gexecute.exe

```
Welcome to Critter Farm

My Critter's name is Poochie

The animals in the farm are:
Larry
Moe
Curly
Groucho

Bye-bye.
```

The program first adds one critter with the name Poochie, directly accessing the Critter class. Then it builds a farm of four critters, with the names shown, and then displays the names of the critters in the farm. What I am doing is using this to show how we can build a farm object out of critter objects, and the farm object in turn can be built using the container library vector component. This shows how Ada objects and structures can be freely mixed, building the farm object from a vector object of critter objects, each of which holds a string object for the critter name.

Here is the new Critter object, which I have built in a new package called CritterNamePak.

In this package I introduce a new keyword, Access, which I use to create a reference to the Critter object, which I will then use to create critters when the program runs. This is called dynamic object instantiation, where new object instances can be created when the program runs, as against static instantiation, where a fixed number of object instances are built when the program is compiled. The next section will discuss access types in greater detail.

Spec:

```
with ada.strings.Unbounded;
USE Ada.Strings.Unbounded;

PACKAGE CritterNamePak IS

   TYPE Critter IS PRIVATE;
   TYPE CritterRef IS ACCESS
       Critter;

   PROCEDURE Init (Crit : IN OUT
       Critter; Name : String);
   Function GetName(Crit :
       Critter) return string;

PRIVATE
   TYPE Critter IS
     record
       Name : Unbounded_String;
    END Record;

END CritterNamePak;
```

Body:

```
PACKAGE BODY CritterNamePak IS

   FUNCTION GetName(Crit :
       Critter) Return String IS
   BEGIN
      RETURN to_string(Crit.Name);
   END;

   PROCEDURE Init(Crit : in out
       Critter; Name : String) IS
   BEGIN
       Crit.Name :=
        to_unbounded_string(Name);
   END;

END CritterNamePak;
```

I have built this critter object to contain just a single item, a string object holding the critter's name. Since I have made it Private, I need a procedure Init to place the actual name into the string, and a function GetName to report the name of the Critter. Since the name is held in a string object, this shows that one way to use aggregation when you're defining a class is to declare a data member that can hold another object. That's what I did in Critter, where I declare the data member to hold an unbounded string object: Name : Unbounded_String;. I made the names unbounded as I don't know in advance how long each name will be. This means that when I get an actual name as a simple string, I must convert it to unbounded, and when reading it back for display, I convert it back. I also define a reference to the critter's name using **`TYPE CritterRef IS ACCESS Critter;`** Access types allow dynamic creation of objects and will be discussed in more detail in the next section.

Generally, you use aggregation when an object includes another object. In this case, a critter has a name. These kinds of relationships are called *has-a* relationships (each critter has a name).

Next, I define the Critter Farm in package CritterFarmPak. As I mentioned above, I will create the farm using a vector container. This shows how I can use containers to hold the data members for objects. The single data member I declare for the class is simply a vector called Critter_Container that holds Critter objects. Here is the object spec and body:

Spec:

```
WITH
Ada.Containers.Indefinite_Vectors;
USE Ada.Containers;
```

```
WITH CritterNamePak; USE
CritterNamePak;

PACKAGE CritterFarmPak IS
```

```
   PACKAGE Critter_Container IS
     NEW Indefinite_Vectors
        (Natural, Critter);
   USE Critter_Container;

   PROCEDURE add (Farm : IN OUT
       Vector; myName : string);
   PROCEDURE CallRoll(Farm :
       Vector);

END CritterFarmPak;
```

Body:

```
WITH Text_Io; USE Text_Io;

PACKAGE BODY CritterFarmPak IS
```

```
PROCEDURE add (Farm : IN OUT
       Vector; myName : string) is
      crit : critter;
  BEGIN
    init(Crit, myName);
    Farm.Append (Crit);
  END;

  PROCEDURE CallRoll(Farm :
      Vector) IS
  BEGIN
    put_line("The animals in the
           farm are:");
    FOR eachCrit OF Farm LOOP
      Put_Line(getName(eachCrit));
    END LOOP;
  END;

END CritterFarmPak;
```

As described, a farm is built as a vector of critters. I have created two subprograms, a procedure, add, which accepts two arguments, a Farm and a String and makes a new critter, Crit, initialises it with the string using the procedure Init from CritterNamePak for the name of the critter, then appends the new critter to the farm using the procedure Append from the vector container package. The CallRoll procedure lists the names of all the critters in the farm by looping through the farm with the statement FOR eachCrit OF Farm LOOP, using getName to get the name of eachCrit and display it with Put_Line. The main procedure will then add critters to the farm with the add procedure, and then list the names of the critters in the farm. Here is the main program:

```
WITH CritterNamePak; USE CritterNamePak;
With CritterFarmPak; Use CritterFarmPak;
WITH Text_IO; USE Text_Io;

PROCEDURE CritterFarm IS
   Crit : CritterRef;
   Farm : Critter_Container.vector;  -- create an empty farm vector

BEGIN
   Put_Line("Welcome to Critter Farm"); New_Line;
   Crit := NEW Critter;          -- create a single critter
   Init(Crit.all, "Poochie");    -- and call it Poochie
   Put("My Critter's name is ");
   Put_Line(GetName(Crit.All));  -- display its name
   New_Line;

   Add(Farm, "Larry");        -- add four critters to the farm
   Add(Farm, "Moe");
   Add(Farm, "Curly");
   Add(Farm, "Groucho");

   CallRoll(Farm);

   new_line;
   Put("Bye-bye.");
END CritterFarm;
```

In this example, I have also introduced a new feature, the ability to create objects dynamically, using Access Types and this will be discussed in the next section.

12 Access types

Ada supports a strong model of memory management and dynamic programming, using Access Types. All programming languages support addressing, whereby data is accessed by address rather than by its value. In Ada, this is done using Access Types, which are known in other languages as references or pointers.

In Ada, access types are used for setting up dynamic data structures and for some operations in object oriented programming. In these cases, the data cannot always be set up before the program is run. For simple applications, they are not used and for many data structures, to use the container library, described in Ch 4, is the best approach. For those coming from a C or C++ background, where pointers and references are heavily used, the limited use of access types in Ada may seem strange. In other languages, references are often used for parameter passing to subprograms, whereas in Ada the form of parameter passing is usually left to the Ada compiler system.

Access type variable designate (point to) values of other types (usually user-defined composite types). In the Critter Farm example above, I defined an access type in the CritterNamePak as:

```
TYPE CritterRef IS ACCESS Critter;
```

Then in the main procedure I declared an access variable Crit as

```
Crit : CritterRef;
```

This allows me to write, in the main body,

```
Crit := NEW Critter;
```

which creates a new critter object when the program runs and allocates some memory for it.

All objects of this type are deallocated when the scope of the access type is exited, which depends on just where it was created.

So far, whenever you've declared a variable, Ada has allocated the necessary memory for it. When the subprogram in which that the variable was created, ended, Ada freed the memory. This memory, which is used for local variables, is called the stack. But there's another kind of memory that persists independent of the functions in a program. You, the programmer, are in charge of allocating and freeing this memory, collectively called the storage pool (a memory storage area for variables of this type).

At this point, you might be thinking, "Why bother with another type of memory? The stack works just fine, thank you." Using the dynamic memory offers great benefits that can be summed up in one word: efficiency. By using dynamic memory, you use only the amount of memory you need at any given time. If you have a game with a level that has 100 enemies, you can allocate the memory for the enemies at the beginning of the level and free the memory at the end. The storage pool mechanism also allows you to create an object in one function that you can access even after that function ends. You might create a screen object in one function and return access to it. In fact, for advanced extensible objects, the use of access types and dynamic memory is essential, so you'll find that dynamic memory is an important tool in writing any significant game.

Any kind of variable, object or function in Ada can be referenced by an access type. Here is a simple program that shows access to an integer type:

```
-- Referencing : Demonstrates using Access types

WITH Text_Io; USE Text_Io;
WITH Ada.Integer_Text_IO; USE Ada.Integer_Text_IO;

PROCEDURE AccessDemo IS
   MyScore : ALIASED Integer := 1000;
     -- MyScore is accessed directly and by access type, so must be aliased
   TYPE ScoreType IS ACCESS ALL Integer;  -- create an integer access type
   MikesScore : ScoreType := MyScore'ACCESS;  -- points to MyScore
   HarrysScore : ScoreType;
BEGIN

   Put ("MyScore is: "); Put(MyScore); New_Line;
   Put ("MikesScore is: "); Put(MikesScore.All); New_Line(2);

   Put_Line( "Adding 500 to myScore");
   MyScore := MyScore + 500;
   Put( "MyScore is: "); Put( MyScore); New_Line;
   Put( "MikesScore is: "); Put( MikesScore.All); New_Line(2);
   Put_Line( "Adding 500 to mikesScore");
   MikesScore.All := MikesScore.All + 500;
   Put( "MyScore is: "); Put(MyScore); New_Line;
   Put( "MikesScore is: "); Put(MikesScore.All); New_Line(2);
   Put_Line( "Create HarrysScore, make it = to MikesScore + 500");
   HarrysScore := NEW Integer;  -- make a new integer dynamically in pool
   HarrysScore.All := MikesScore.all + 500;
   Put( "HarrysScore is: "); Put(HarrysScore.All); New_Line(2);
   Put( "MikesScore is still: "); Put(MikesScore.All); New_Line(2);
   Put_Line("Now make MikesScore point to HarrysScore");
   MikesScore := HarrysScore;
   Put( "HarrysScore is: "); Put(HarrysScore.All); New_Line(2);
   Put( "MikesScore is now: "); Put(MikesScore.All);

END;
```

When you run this program, you will see that MyScore and MikesScore always hold the same value, but HarrysScore holds a separate value and so can be different from MyScore and MikesScore. Note that the .all modifier dereferences the variable, which means it accesses the value rather than the reference. So, MikesScore is a reference (an access type) which points to an integer object while MikesScore.all is the value stored in the integer object pointed to.

The keyword NEW is called an Allocator. It creates an object and returns an access value that designates (points to) the object. So, HarrysScore := NEW Integer; creates a new integer object in the pool by setting aside the required memory space for it and then returns an access value which I store in the access variable HarrysScore. I then store a different value in the integer in the pool and display it.

Finally, I make MikesScore point to HarrysScore and display the values pointed to by both access variables, which now results in both displaying the same value.

This kind of simple use of access types is never used in Ada as Ada does not need them. As mentioned above, access types are needed for access to some types of objects, as we will see, are heavily used in many of the library packages and are also needed for interfacing to libraries written in other languages, such as graphics libraries for windows or linux.

Deleting objects from the pool

Ada automatically deletes objects from the pool whenever the scope of the access type is exited so there is normally no need to delete them, unlike in C++, for example. If you need the object to persist, the access type needs to be defined globally, as for any other global variable.

If you need to delete an object from the pool, this is done using the Ada unchecked deallocation library package, which deletes the object from the pool and frees the memory allocated to it.

Dangling references

In most languages with pointers, a source of many programming errors is the problem of dangling references, which are pointers that either point to nothing or point to invalid data. In Ada, access types are subject of so-called accessibility checks, which in Ada are intended to prevent appearance of dangling pointers. Though accessibility checks can be circumvented by using unchecked_access, a wise programmer will limit its use!

Working with Data Members and the Pool

You've seen how you can use aggregation to declare objects that are composed of other objects. You can also declare object data members that are pointers to values or other objects in the pool. You might want to do this if you are building up a large 3D scene only have access to the 3D scene through a pointer. You can also write member subprograms to access such objects an implement whatever behaviours you need.

12.1 Word Jumble Game revisited

Container types carry some overhead, so let's revisit the Word Jumble game from chapter 5 using access types instead of containers. Recall that Word Jumble is a puzzle game in which the computer picks a word at random from a list, then creates a jumbled up word with the letters in random order. The player must attempt to guess the word and may ask for a hint if stuck.

We will store the words and hints in two sets of strings. We immediately face a problem in Ada in that array elements must all be of the same type and size, whereas our strings all vary in length. So why try to use an array at all? We do this because it is a convenient way to set up our lists of words and hints so one can be chosen at random, using a random number to set the array index.

So how do we overcome this problem? The solution is to use the access attribute for the strings, and store those in the array. The access attribute for any object or item designates the object or item and provides access to it. Think of it as a kind of map reference to the item. For our purposes, the access attribute values designating the strings are all of equal size, so can be placed as variables in an array, while the strings themselves vary in length.

Since the name of each string refers to it, and so does the access variable that designates it, these are aliases for the string, and so need to be marked as aliases. This is a bit like a person who has a name and a nickname, called an alias.

Here is the program

```
-- Word Jumble using an array of access types
-- The classic word jumble game where the player can ask for a hint

WITH Text_IO; USE Text_IO;
WITH Ada.Strings.Unbounded; USE Ada.Strings.Unbounded;
WITH Ada.Numerics.Discrete_Random;
WITH Ada.Strings; USE Ada.Strings;
WITH Ada.Text_IO.Unbounded_IO; USE Ada.Text_IO.Unbounded_IO;

PROCEDURE Jumble1 IS
   NUM_WORDS : CONSTANT := 6;

   -- word and hints -- ALIASED says each item will have a second reference
   Str1  : ALIASED String := "wall";
   Hint1 : ALIASED String := "Banging your head against something?";
   Str2  : ALIASED String := "glasses";
   Hint2 : ALIASED String := "These might help you see the answer.";
   Str3  : ALIASED String := "labored";
   Hint3 : ALIASED String := "Going slowly, is it?";
   Str4  : ALIASED String := "persistent";
   Hint4 : ALIASED String := "Keep at it.";
   Str5  : ALIASED String := "jumble";
   Hint5 : ALIASED String := "It's what the game is all about.";
   Str6  : ALIASED String := "lecture";
   Hint6 : ALIASED String := "Lots of boring words and powerpoints.";

   TYPE StrAcc IS ACCESS ALL String;
   TYPE StringsA IS ARRAY (0 .. NUM_WORDS - 1, 0 .. 1) OF StrAcc;
   Words : StringsA :=
      ((Str1'ACCESS, Hint1'ACCESS),  -- Initialise array elements to
       (Str2'ACCESS, Hint2'ACCESS),  -- to access the words and hints
       (Str3'ACCESS, Hint3'ACCESS),
       (Str4'ACCESS, Hint4'ACCESS),
       (Str5'ACCESS, Hint5'ACCESS),
       (Str6'ACCESS, Hint6'ACCESS));

   PACKAGE RInt IS NEW Ada.Numerics.Discrete_Random (Integer);
   USE RInt;
   G      : Generator;
   Choice : Integer;
   Guess  : Unbounded_String;  -- guesses can vary in length
   Temp   : Character;
   Index1 : Integer;
   Index2 : Integer;

BEGIN
   Put_Line("      Welcome to Word Jumble!"); New_Line;
   Put_Line("Unscramble the letters to make a word.");
   Put_Line("Enter 'hint' for a hint.");
   Put_Line("Enter 'quit' to quit the game."); New_Line;

   Reset(G);
   Choice := Random(G) Mod NUM_WORDS; -- Choice range is 0 .. NUM_WORDS - 1
   DECLARE   -- Start a new block of code, including declarations
      TheWord : String := WORDS (Choice, 0).ALL;  -- .ALL says copy the
      TheHint : String := WORDS (Choice, 1).ALL;  -- whole string
      Jumble : String  := TheWord;
      Len     : Integer := TheWord'Length;
   BEGIN
      FOR I IN 1..Len LOOP  -- jumble up the word by swapping letters
         BEGIN
            Index1 := (Random(G) Mod Len) + 1; -- string index range
            Index2 := (Random(G) Mod Len) + 1; -- is 1 .. length
            Temp := Jumble(Index1);        -- swap randomly chosen letters
            Jumble(Index1) := Jumble(Index2);
            Jumble(Index2) := Temp;
         END;
      END LOOP;
      Put("The jumble is: ");
      Put(Jumble); New_Line; New_Line;

-- Now get and check the guesses. Player may ask for a hint, or quit

      Put("Your guess: ");
      Get_Line(Guess);
      WHILE ((Guess /= TheWord) AND (Guess /= "quit")) LOOP
         BEGIN
            IF (Guess = "hint") THEN
               Put_Line(TheHint);
            ELSE
               Put_Line("Sorry, that's not it.");
            END IF;
            Put("Your guess: ");
            Get_Line(Guess);
         END;
      END LOOP;
      IF (Guess = TheWord) THEN
         BEGIN
            New_Line;
            Put_Line("That's it!  You guessed it!");
         END;
      END IF;
      New_Line;
      Put_Line("Thanks for playing.");
   END;
END;
```

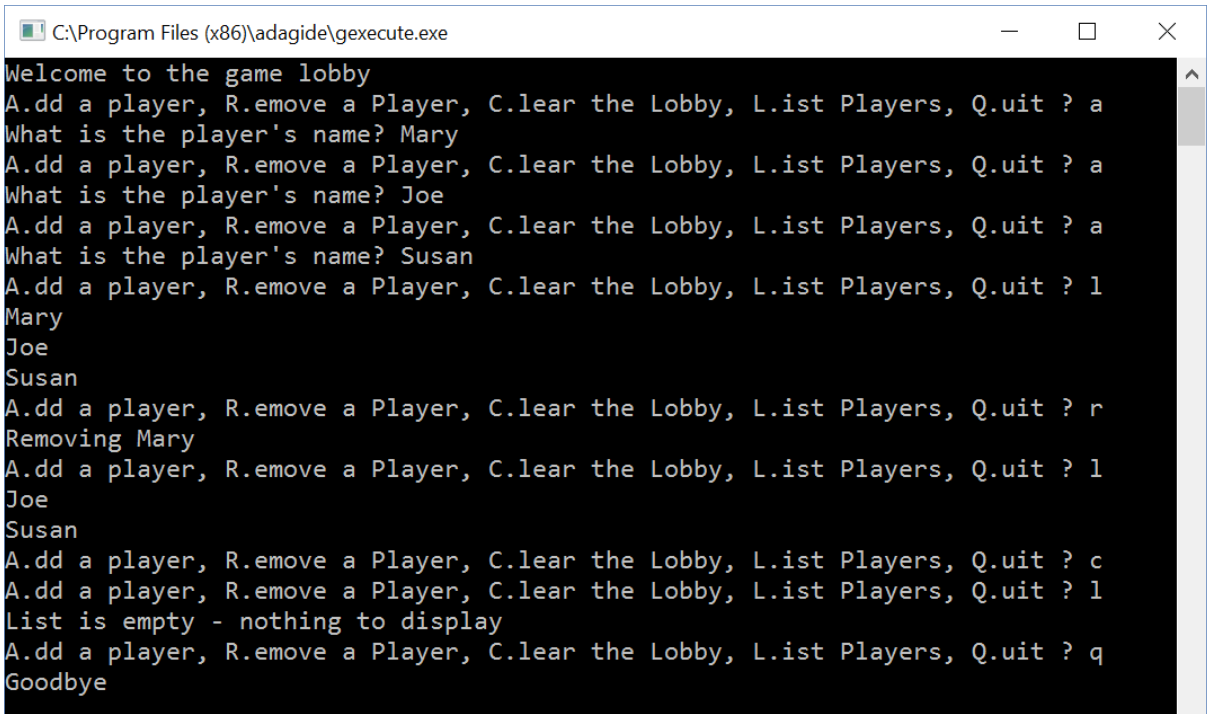
The rest of the code is the same as the example in Ch 5.

This program has introduced several new concepts in using access attributes, so should be studied closely. We will use access types much more later.

12.2 Introducing the Game Lobby Program

Now that we can declare access types and know a bit about how to use them, let's look at basic operations for access values with the Game Lobby program.

The Game Lobby program simulates a game lobby—a waiting area for players, usually in an online game. The program doesn’t actually involve an online component. It creates a single line in which players can wait. The user of the program runs the simulation and has four choices: to add a person to the lobby, remove a person from the lobby (the first person in line is the first to leave), clear out the lobby, or quit the simulation. Here is the program in action:



```
Welcome to the game lobby
A.dd a player, R.emove a Player, C.lear the Lobby, L.ist Players, Q.uit ? a
What is the player's name? Mary
A.dd a player, R.emove a Player, C.lear the Lobby, L.ist Players, Q.uit ? a
What is the player's name? Joe
A.dd a player, R.emove a Player, C.lear the Lobby, L.ist Players, Q.uit ? a
What is the player's name? Susan
A.dd a player, R.emove a Player, C.lear the Lobby, L.ist Players, Q.uit ? l
Mary
Joe
Susan
A.dd a player, R.emove a Player, C.lear the Lobby, L.ist Players, Q.uit ? r
Removing Mary
A.dd a player, R.emove a Player, C.lear the Lobby, L.ist Players, Q.uit ? l
Joe
Susan
A.dd a player, R.emove a Player, C.lear the Lobby, L.ist Players, Q.uit ? c
A.dd a player, R.emove a Player, C.lear the Lobby, L.ist Players, Q.uit ? l
List is empty - nothing to display
A.dd a player, R.emove a Player, C.lear the Lobby, L.ist Players, Q.uit ? q
Goodbye
```

The Player Class

The first thing I do is create a Player class to represent the players who are waiting in the game lobby. Because I don't know how many players I'll have in my lobby at one time, it makes sense to use a dynamic data structure. Normally, I'd go to my toolbox of containers from the container library. But I decided to take a different approach in this program and create my own kind of container using dynamically allocated memory that I manage. I didn't do this because it's a better programming choice—always see whether you can leverage good work done by other programmers, like the container library—but because it makes for a better game programming example. It's a great way to really see dynamic memory in action and to see how a basic data structure works.

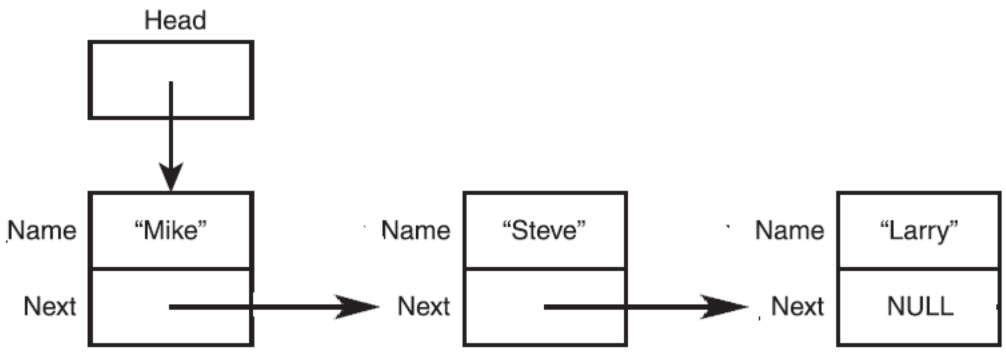
As I have done before, I start by creating a package containing the player object and its member data and subprograms. We will want to be able to add a player to the game lobby, remove a player (first in – first out), display the players in the lobby and clear the lobby, so this defines four subprograms we will need. The specification of the code is

```
PACKAGE LobbyPak IS  -- package definition -  define a new type, Player

  TYPE WordType IS String(1..20); -- names can be a maximum of 20 chars

  TYPE Players IS PRIVATE;

  PROCEDURE Add (L: IN OUT Players);  -- Add new player to end of Lobby
  PROCEDURE Remove(L: IN out Players); -- Remove first player in the Lobby
  PROCEDURE Display (L: IN Players);  -- Displays all players in the Lobby
  PROCEDURE Clear (L: IN OUT Players); -- Clear the whole lobby
PRIVATE
```

```
  TYPE PlayerNode;
  TYPE Players IS ACCESS PlayerNode;
  TYPE PlayerNode IS RECORD
    Name: WordType;
    Next: Players;
  END RECORD;

END LobbyPak;
```

For this example I will allow a maximum length of 20 characters for the players' names. Keeping the records a fixed size simplifies the code.

The PlayerNode class above now has two data members: the Name data member holds the name of a player. That's pretty straightforward, but you might be wondering about the other data member, Next. It's a pointer to a Player object, which means that each Player object can hold a name and can point to another of the Player objects. This will allow me to string together a list of players, linked by means of the pointers. This is called a *linked list* and is a very common and useful structure in games



programming. Individual elements of linked lists are often called nodes. The diagram provides a visual representation of a game lobby—a series of player nodes linked with one player at the head of the line and the last player having a NULL link:

One way to think about the player nodes is as a group of train cars that carry cargo and are connected. In this case, the train cars carry a name as cargo and couplings linking them are the pointer data members, Next. Each time a new player is created with NEW PlayerNode, Ada allocates memory from the pool for the new Player instance to be added to the list. I will also create a header pointer for the list, which I will call Lobby, in the main procedure. The Lobby pointer links to and provides access to the first Player object at the head of the list.

When a player is removed from the list, or when the lobby is cleared, the memory in the pool allocated to the player, or to the lobby, should be freed. To do this, I need to 'with' a package called unchecked_deallocation and make an instance of it for the Players class. It also needs the compiler pragma controlled (a program is an instruction to the compiler, in this case to build a deallocator for the Players type). I will give the deallocator the name Free, which is a common usage.

Now I must set up the body of the player class to implement the member subprograms in the package body. Here is the code

```
WITH Text_Io;USE  Text_IO;
WITH Unchecked_Deallocation;
```

```
PACKAGE BODY LobbyPak IS  --  Player implementation

   PROCEDURE Free IS
   NEW Unchecked_Deallocation(PlayerNode, Players);
   PRAGMA Controlled(Players);

   PROCEDURE Add(L : IN OUT Players) IS
      Current : Players; -- designates each node of input list in turn
   BEGIN -- Add
      Put("What is the player's name? ");
      declare
         NewName : constant String := Get_Line; -- get players name
         Name : WordType := (OTHERS => ' ');
         Len : constant integer := integer'Min(NewName'Length,20);
      BEGIN
         Name(1..Len):= NewName(1..Len); -- must <= 20 chars
      IF L = NULL THEN
         L := NEW PlayerNode'(Name,NULL);
      ELSE
         Current := L;  -- initialize the loop – start at the head
         -- then search until the end
         WHILE Current.Next /= NULL LOOP
            Current := Current.Next;
         END LOOP;
         -- we found the end; Current designates last node
         -- so attach a new node to the node Current designates
         Current.Next := NEW PlayerNode'(Name, NULL);
         END IF;
      END;
   END Add;

   PROCEDURE Display (L : IN Players) IS
      Current : Players; -- designates each node of input list in turn
   BEGIN -- Display
      IF L = NULL THEN
         Put_Line("List is empty - nothing to display");
      ELSE
         -- initialize the loop
         Current := L;
         -- search until the end
         LOOP
            Put_Line(Current.Name);
            exit when Current.Next = NULL;
            Current := Current.Next;
         END LOOP;
      END IF;
   END Display;

   PROCEDURE Remove(L : IN OUT Players) IS
      R : Players;
      BEGIN
         IF L = NULL THEN
            Put_Line("Lobby is empty, no one to remove");
         ELSE
            Put_Line("Removing " & L.Name);
            R := L;
            L := L.Next;
            Free(R);  -- release memory from pool
         END IF;
      END;
      PROCEDURE Clear (L: IN OUT Players) IS
      BEGIN
         Free(L);
      END;

   END LobbyPak;
```

I have chosen to implement Add iteratively. First I check if there no players in the game lobby, in which case the new player becomes the first in the list. If there are already players in the lobby, I have to first search to find the end of the list of players, then add the new player to the end by replacing the NULL empty link of the old last player with a link to the new player. I have used some neat programming tricks provided by Ada to do this, but first I have to read in the new name, which must be exactly 20 characters long, so I read in the name and then paste it into a 20 character blank string using the length attribute of the string holding the new name, making sure I paste in 20 characters or less, or I might get a run-time constraint error.

Then I check the lobby header pointer, which must be passed as a parameter to Add. If it is NULL, the lobby is empty, so I create a new player, give the player the name I just constructed and set L, the lobby header pointer, to point to it. Note that Ada allows me to initialise the new player with the statement L := NEW PlayerNode'(Name,NULL); where I create a new player by calling NEW and plug in the member data fields of the newly allocated block with a record aggregate (Word, NULL). The apostrophe preceding the aggregate is required; the construct PlayerNode'(Word,NULL) is called a qualified aggregate. The Ada Language Reference Manual (LRM) section 4.3.1 give several examples of aggregates used to initialise records.

Next, if the Lobby is not empty, I have to find the end of the list of players. I do this by moving through the list one node at a time, starting from the head of the list with current := L; then moving from node to node with Current := L; then searching node by node with Current := Current.Next; until the end, where the last node has a NULL in its Next data member. The While loop will search until it hits the NULL and then exits with Current pointing to the last node in the list of players. Then I make that node point to the new Player object, with Current.Next := NEW PlayerNode'(Name, NULL); which has the effect of adding the new object to the end of the list.

T r a p: Add marches through the entire list of Player objects every time it's called. For small lists this isn't a problem, but with large lists this inefficient process can become unwieldy. There are more efficient ways to do what this function does. As an exercise, add a tail pointer which always points to the end of the list.

The Display member procedure is very similar to Add, but since it does not change the list, its parameter has mode IN. Again, it first checks if the list is empty. If not, it scans through the list, displaying the name in each node, then exits the loop `when Current.Next = NULL;` The structure is a bit simpler than Add as I do not have to add a new player to the list.

The Remove member procedure removes the player at the head of the line. It first tests the list head and if it's NULL, then the lobby is empty and I display a message that says so. Otherwise, the first player object in the list is removed. The function accomplishes this by creating a pointer, R, and pointing it to the first Player object in the list with R := L;. Then the function sets the Lobby head to the next player in the list or NULL with L := L.Next;. Finally, I destroy the old first player object pointed to by R with Free(R); where Free was instantiated from the unchecked_deallocation package for objects of type Players.

Finally, the Clear procedure simply uses Free(L) to free the lobby.

As I mentioned above, entering a name longer than 20 characters in Add will result in the name being truncated to 20 characters. To deal with this, as an exercise, tell the user, when the new name is entered, to keep it to 20 characters or less and add a test to warn the user if the name entered is longer than 20 characters. Alternatively, redesign the package to use unbounded strings, best done using unbounded lists from the container library, in which case the names can be any length.

The game lobby main program again uses a simple menu structure to call the subprograms provided in the lobby package. It is similar to the player's database we looked at previously, but manages the players as a waiting queue rather than an array of players.

```
WITH Text_IO; USE Text_IO;
with LobbyPak; use LobbyPak;

PROCEDURE GameLobby IS
   Lobby : Players;
   Choice   : Character;

BEGIN
   Put_Line("Welcome to the game lobby");
   LOOP
      Put( "A.dd a player, R.emove a Player, C.lear the Lobby, L.ist 
Players, Q.uit ? " );
      Get( Choice );
      Skip_Line;
      CASE Choice IS
         WHEN 'A' | 'a' =>  Add(Lobby);
         WHEN 'R' | 'r' =>  Remove(Lobby);
         WHEN 'C' | 'c' =>  Clear(Lobby);
         WHEN 'L' | 'l' =>  Display(Lobby);
         WHEN 'Q' | 'q' =>  EXIT;
         WHEN OTHERS    =>
            Put_Line( "Invalid choice. Please try again." );
      END  CASE;
   END LOOP;
   Put_Line("Goodbye");
END;
```

12.3 More advanced Data Structures

You can build more complex data structures than the simple list we used above, by storing more than one access value in a node. For example, a binary tree is a set of nodes, where each node has three links, a way to locate its "parent" node, its "left child" node, and its "right child" node. Each node also has data it contains. The container library contains a variety of trees, lists, queues, sets, maps, etc. Advanced games programmers will use many such structures.

The Ada container library provides powerful data structuring capabilities. Use them to model the data as closely as possible. It is possible to group logically related data and let the language control the abstraction and operations on the data rather than requiring the programmer or maintainer to do so. Data can also be organized in a building block fashion. In addition to showing how a data structure is organized (and possibly giving the reader an indication as to why it was organized that way), creating the data structure from smaller components allows those components to be reused. Using the features that Ada provides can increase the maintainability of your code. There are many books and online articles about Ada data structures and in the next two chapters we will explore some of them.

12.4 Dangling references

A potential problem when using access types is that of dangling references. This is a source of many programming errors in all languages. Ada tries to minimise the problem by making limited use of access types, nevertheless, the problem still exists. For example, this code shows how a dangling reference might be created:

```
P1 := new Object;
P2 := new Object;
P2 := P1;
```

Since there is no intervening assignment of the value of P2 to any other access variable, the object created on the second line is no longer accessible after the third line. The only reference to the allocated object P2 has been dropped and there is no way to access the object. Of course, you would not do anything so silly, but in a long program that manipulates references in data structures like lists and trees, it is all too easy to accidently create a dangling reference.

Another problem arises with dynamically allocated objects. These are objects created by the execution of an allocator (new). Allocated objects referenced by access variables allow you to generate aliases, as we have seen, which are multiple references to the same object. Anomalous behaviour can arise when you reference a deallocated object by another name.

For these reasons, it is a good idea wherever possible to use the well designed and thoroughly tested components in the container library.

13 Generic Packages

Ada extends the concepts discussed in chapters 10 and 11 above by implementing generic packages. A generic unit is a template for a subprogram or a package with some items not yet defined, so it cannot be executed directly. A copy of the generic unit can be instantiated, that is, a copy is made with the undefined items completed, so that the resulting unit can be executed just as any other subprogram or package. The key idea is that of Reusability. Reusable parts often need to be changed before they can be used in a specific application. They should be structured so that change is easy and as localized as possible. One way of achieving adaptability is to use Ada's generic construct to produce parts that can be appropriately instantiated with different parameters. This approach avoids the error-prone process of adapting a part by changing its code and successful games programmers will fully exploit this capability by building up their own libraries of generic code.

Anticipated changes, that is, changes that can be reasonably foreseen by the developer of the part, should be provided for as far as possible. Unanticipated changes can only be accommodated by carefully structuring a part to be adaptable. Many of the considerations pertaining to maintainability apply. If the code is of high quality, clear, and conforms to well-established design principles such as information hiding, it is easier to adapt in unforeseen ways. For excellent, detailed guidelines on how to do this, see http://www.adaic.org/resources/add_content/docs/95style/html/sec_8/8-3.html.

The Ada programming system itself makes heavy use of generic library packages, and we have already used them in our code. For example Ada.Text_IO provides generic packages that allow you to create instances for text_io of integers, reals, strings, etc. This illustrates a key feature of generic packages, that they should be adaptable and extendable by the person reusing them.

To kick off, here is the basic idea for a generic package to swap two items. Swap is one of the core procedures for sorting, and in games is also used for things like fighters swapping weapons. So we see we might want to swap integers, or reals, or strings, or weapons. We'd like to be able to write a general swap procedure that would swap any two items.

The spec and body for such a generic swap routine are written as:

Spec:

```
-- Here's the spec

GENERIC
   TYPE Item IS PRIVATE;
      -- item can be anything

PACKAGE Gen_Swap IS
   PROCEDURE Swap (
    Left, Right : IN OUT Item);

END Gen_Swap;
```

Body

```
-- .. and here's the body:

PACKAGE BODY Gen_Swap IS

   PROCEDURE Swap (
     Left, Right : IN OUT Item) IS
     Temp : Item;
   BEGIN
     Temp := Left;
     Left := Right;
     Right := Temp;
   END Swap;

END Gen_Swap;
```

The idea is that the generic parameter Item can be replaced with any valid type.

When you create a generic version of a subprogram or package, you write the subprogram as you normally would, but using a few generically-named types. Then start the subprogram or package spec with the keyword ``generic'' and a list of the information you'd like to make generic, in this case, Item. The items in this list are called the generic formal parameters and here I have only one; this list is similar to the list of parameters in a procedure declaration. These can be types, ranges, values, etc. – anything valid for the code you are writing. Private here means that we don't know yet what type Item is – it can be any valid type and we'll supply that information later.

To use a generic subprogram or package, we have to create a real subprogram or package from the generic version. This process is called instantiating, and the result is called an instantiation or instance. These are big words for a simple concept, which is just to replace the formal parameters with actual types. For example, here's how to create three Swap procedure instances from the generic one:

```
package Swap_Ints is new Gen_Swap(Integer); USE Swap_Ints;
package Swap_Weapons is new Gen_Swap(Weapons); use Swap_Weapons;
package Swap_Strings is new Gen_Swap(String); use Swap_Strings;
```

In all three cases, you then call the procedure swap in the body of the generic as swap(X,Y) where X and Y are the parameters in a procedure call.

So, from here on, you can call the Swap procedure that takes Integers, the Swap procedure that takes Floats, and the Swap procedure that takes Strings. Thus if A and B are both of type Integer, Swap(A,B) would swap their values. As for any other Ada subprogram, if different subprograms share the same name, Ada will determine at compile time which one to call based on the argument types.

Here is an example of how this is done:

```
with Ada.Text_IO, Gen_Swap;
use Ada.Text_IO;
procedure TestGenSwap is
TYPE Weapons IS (sword, laser, M16);
package Swap_Ints is new Gen_Swap(Integer); USE Swap_Ints;
package Swap_Weapons is new Gen_Swap(Weapons); use Swap_Weapons;

   Value1 : Integer := 5;
   Value2 : Integer := 3;
TYPE Arsenal IS ARRAY(integer range <>) OF Weapons;
   MyArsenal : Arsenal := (Sword, Sword, Laser, Laser, M16, M16, M16);
BEGIN
   Put_Line("Integer values are");
   Put_Line(Integer'Image(Value1));
   Put_Line(Integer'Image(Value2));
   Put_Line("Now swap the integers");
   swap(value1, value2);
   Put_Line(Integer'Image(Value1));
   Put_Line(Integer'Image(Value2));
   New_Line;

   Put_Line("Now display the arsenal");
   FOR I IN MyArsenal'RANGE LOOP
      Put(Weapons'image(MyArsenal(I))); Put(" ");
   END LOOP;
   New_Line;
   Put_Line("Now swap the first and last arsenal items");
   Swap(MyArsenal(MyArsenal'First), MyArsenal(MyArsenal'last));
   FOR I IN MyArsenal'RANGE LOOP
      Put(Weapons'Image(MyArsenal(I))); Put(" ");
   END LOOP;
end testGenSwap;
```

In this example, I build two instances of the generic swap package, one for integers called Swap_Ints and the other for Weapons, called Swap_Weapons, where Weapons is an enumeration type I have defined. Next I make two integer variables, Value1 and Value2 and give them values of 3 and 5. Then I define an array type of Weapons, called Arsenal, and make an instance, MyArsenal , that holds a set of weapons. When the program runs, it displays the two values, calls Swap_Int, then displays the two values again to show they were swapped. Then it displays the set of weapons, swaps two of them and displays them again to show they too were swapped.

This example shows how to set up and use a simple generic, which can then be used for any item, including elements of an array as shown here, or for records or other structures, including objects you have defined yourself.

This way of providing instances of a generic is called Static (or Parameterised) Polymorphism. Polymorphism is a means of factoring out the differences among programs, such that programs may be written in terms of their common properties. Static polymorphism is provided through the generic parameter mechanism whereby a generic unit may be instantiated at compile time with any type from a class of types.

Now let’s look at other kinds of generic parameters. Generic formal parameters are the stuff right after the keyword "generic" and here are the things that can be included in a generic formal parameter list:

- Values or variables of any type. These are called `formal objects'. For example, a maximum size might be a useful formal object.
- Any type. These are called `formal types'.
- Packages which are instances of other generic packages. These are called `formal packages'.

Formal types specify the name of a type and what ``kind of type'' is required. A formal type declaration specifies the "minimum" or "worst case" kind of type that is required. The most minimal type in Ada is called a "limited private" type. This is the "worst case" because the keyword "private" means that you may not know anything about how it's implemented, and "limited" means that there might not be assignment or equality operations defined for it. Here’re some examples, with their meanings written beside them:

- type Item is limited private; -- Item can be any type.
- Type Item is private;   -- Item can be any type that has assignment (:=) and equal-to (=) operation.
- type Item is tagged limited private; -- Item can be any tagged type.
- type Item is tagged private;   -- Item can be any tagged type with :=.
- type Item is (<>);   -- Item can be any discrete type, including Character, Integer and Boolean.
- Type item is abstract; -- more about abstract types later.

Let’s look at an example with a generic value.

In many games, we need a game board, and its size depends on the game. Tic-tac-toe uses a 3x3 board, whereas checkers and chess need 8x8 boards. Here are a spec and body for a generic to set up a board with the size as a generic parameter and then a main procedure to draw two boards, one of size 3x3 and one 10x10, with two of the squares marked.

Spec:

```
GENERIC
   Size : Positive;
   Mark_Type : Character;

PACKAGE boardSetupPkg IS
   TYPE Board IS ARRAY
   (1..Size, 1..Size) OF Boolean;

   PROCEDURE DrawBoard(
       Grid_Box : Board);
   PROCEDURE UpdateBox(
       Grid_Box : IN OUT Board;
       Row, Col : Positive);

END boardSetupPkg;
```

Body:

```
WITH Text_Io; USE Text_IO;

PACKAGE BODY BoardSetupPkg IS

  PROCEDURE DrawBoard (
     Grid_Box : IN BOARD) IS
  BEGIN
    New_Line;
    FOR K IN Grid_Box'RANGE LOOP
       Put("--");
    END LOOP;
    Put_Line("-");
    FOR I IN Grid_Box'RANGE LOOP
      FOR J IN Grid_Box'RANGE LOOP
        IF Grid_Box(I,J) THEN
           Put("|");
           Put(Mark_Type);
        ELSE
           Put("| ");
        END IF;
     END LOOP;
     Put("|");
     New_Line;
     FOR K IN Grid_Box'RANGE LOOP
          Put("--");
        END LOOP;
        Put_Line("-");
      END LOOP;
      New_Line;
   END DrawBoard;

   PROCEDURE UpdateBox (
     Grid_Box : IN OUT Board;
       Row, Col : Positive) IS
   BEGIN
      Grid_Box(Row, Col) := True;
   END;

   END BoardSetupPkg;
```

And here is the main procedure

```
with BoardSetupPkg; with text_io;
use text_io;
procedure drawboard is
  X : character;
  PACKAGE Grid3 IS NEW
      BoardSetupPkg (3, '#');
  USE Grid3;
  Package Grid10 is New
      BoardSetupPkg(10, '*');
  USE Grid10;

  SmallBoard : Grid3.Board :=
    (OTHERS => (OTHERS => False));
  BigBoard : Grid10.Board :=
    (OTHERS => (OTHERS => False));
begin
  DrawBoard(SmallBoard);
  Put("About to Update two
        squares in BigBoard. ");
  Put("Hit any key: ");
  get_immediate(X);
  UpdateBox(BigBoard, 3, 5);
  UpdateBox(BigBoard, 7, 2);
  DrawBoard(BigBoard);
end DrawBoard;
```

While this package neatly demonstrates how to set up and use a generic, we can extend it to make it more flexible and reusable. We could, for example, allow non-square boards and the arrangement of different pieces on the board such as O and X for tic-tac-toe, or a set of characters for the pieces in chess. We also like to be able to label the rows and columns. For example, here's a declaration of a chessboard in Ada:

```
  type File is (A,B,C,D,E,F,G,H);
```

```
   type Rank is range 1..8;
   type Square is ... ;           -- some type declaration
   Board : array (File, Rank) of Square;
```

You can now refer to Board(E,5) or Board(G,6) and so on.

For now, as an example of a somewhat different board setup, please take a look at Conway's Life in Life2.adb, along with the associated package, grid2. The original is by E. Burke, then a student at Florida State University (see http://alpha.fdu.edu/~levine/reuse_course/students/index.html). The code uses many of techniques we have learned so far, plus a few new ideas. Feel free to experiment with the code and perhaps add some new starting patterns. This version of Life is limited by the size of the screen and life forms die when they hit the edges, so there is lots to do to improve it.

I have only touched lightly on Generics and there is much more I could say, but this is enough to give you a taste of how they work. The Ada library itself makes heavy use of generics in the containers, the IO subsystems, the numerics packages, etc. and you can find an excellent treatment in Barnes' book, "Programming in Ada 2012". But let's now turn to dynamic polymorphism and inheritance.

14 Dynamic Objects

So far, we have most;y looked at static objects, where everything about the architecture of the object is fixed when the program is compiled and cannot be changed when the program runs. Object oriented programming (OOP) purists often insist that static objects are not true objects and that true OOP cannot be achieved with static objects alone. Ada, along with C++, Java and many other modern languages thus fully supports dynamic objects.

Let's see now how Ada2012 supports full OOP classes with inheritance and polymorphism, user-defined initialization, finalization, and assignment, and multiple inheritance. Two key concepts are:

- Abstract data types (ADTs) where types are defined by only their operations, leaving the type structure hidden
- OOP where types are reused by extending and specializing them

Defining a type by its operations and hiding its data structure protects the integrity of the type implementation from client intrusion and permits changes in implementation without disturbing the clients

ADTs are provided in Ada by private and limited private types. Private parts of the package specification hide the structure of private types. They are defined to support module connections but are hidden from client intrusion,

As we have seen in the examples in chapters 11 and 12, packages and private types provide complete support for

- Encapsulation: private and limited private types define the abstraction through primitive operations
- Derived types provide some support for inheritance
- Operations on types can be inherited, overridden, or added
- But, additional data components cannot be added
- Type overloading and generics provide only static "compile-time" polymorphism.

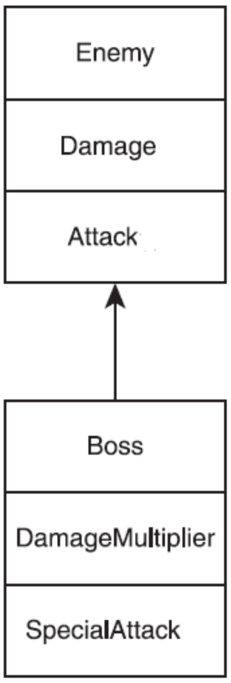
To overcome these limitations, Ada adds a tag to records to provide for extensions to record and private types. We just add the keyword tagged to the record spec:

```
type root_type is
  tagged record
    etc.
```

Tagging a record type permits its reuse through Inheritance, class-wide programming and dynamic (i.e. runtime) dispatching.

All this sounds a bit obscure and theoretical, so let's look at a simple example. Since one of the key elements of OOP is inheritance, tagging allows you to derive a new class from an existing one. When you do so, the new class automatically inherits (or gets) the data members and member functions of an existing class. It's like getting the work that went into the existing class for free!

Inheritance is especially useful when you want to create a more specialized version of an existing class because you can add data members and member functions to the new class to extend it. For example, imagine you have a class Enemy that defines an enemy in a game with a member subprogram Attack and a data member Damage. You can derive a new class Boss from Enemy. This means that Boss could automatically have Attack and Damage without you having to write any code for them at all. Then, to make a boss tougher, you could add a new member subprogram SpecialAttack and a data member DamageMultiplier to the Boss class. Take a look at this diagram, which shows the relationship between the Enemy and Boss classes.



This diagram shows that Boss inherits Attack and Damage from Enemy while defining its own SpecialAttack and DamageMultiplier.

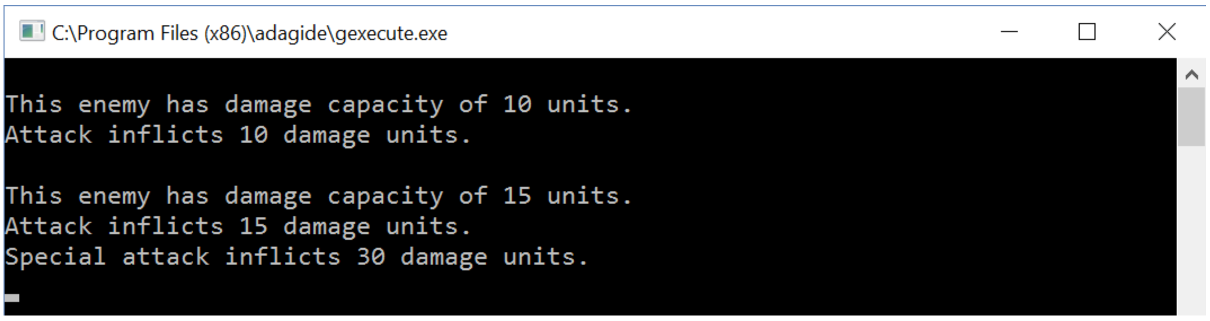
One of the many advantages of inheritance is that you can reuse classes you've already written. This reusability produces benefits that include:

- Less work. There's no need to redefine functionality you already have. Once you have a class that provides the base functionality for other classes, you don't have to write that code again.
- Fewer errors. Once you've got a bug-free class, you can reuse it without errors cropping up in it.
- Cleaner code. Because the functionality of base classes exist only once in a program, you don't have to wade through the same code repeatedly, which makes programs easier to understand and modify.

Most related game entities cry out for inheritance. Whether it's the series of enemies that a player faces, squadrons of military vehicles that a player commands, or an inventory of weapons that a player wields, you can use inheritance to define these groups of game entities in terms of each other, which results in faster and easier programming.

14.1 Introducing the Simple Boss Program

The Simple Boss program demonstrates inheritance. In it, I define a class for lowly enemies, Enemy. From this class, I derive a new class for tough bosses that the player has to face, Boss. Then, I instantiate an Enemy object and call its Attack member function. Next, I instantiate a Boss object. I'm able to call Attack for the Boss object because it inherits the member function from Enemy. Finally, I call the Boss object's SpecialAttack member function, which I defined in Boss, for a special attack. Since I define SpecialAttack in Boss, only Boss objects have access to it. Enemy objects don't have this special attack at their disposal. SpecialAttack also allows the Boss to seize and use the Enemy's weapon, since that is something we expect Bosses to be able to do! Here is the result of running the program:



```
This enemy has damage capacity of 10 units.
Attack inflicts 10 damage units.

This enemy has damage capacity of 15 units.
Attack inflicts 15 damage units.
Special attack inflicts 30 damage units.
```

Here is the EnemyPak class definition (in a package, as usual)

Spec

```
package EnemyPak2 is

   -- Begin with a basic Enemy
type.
   type Enemy is tagged private;

   procedure Attack(
        AnEnemy : Enemy);

   Procedure SetDamage(AnEnemy :
     in out Enemy;
     Damage : integer);

   -- Extend the basic type to a
   -- BOSS type.

   TYPE Boss IS NEW Enemy WITH
     PRIVATE;

   PROCEDURE SetMultiplier(
        TheBoss : in out Boss;
        Multiplier : Integer);

   procedure SpecialAttack(
        TheBoss : Boss;
        anyEnemy : Enemy'class);
-- AnyEnemy here is a class-wide
-- argument, so can be of type
-- Enemy or Boss


-- Display damage can also accept
-- objects of any type within the
-- Enemy class heirarchy.
   procedure DisplayDamage(
        Any_Enemy : Enemy'Class);

private

   type Enemy is tagged
      record
         Damage : INTEGER;
      end record;

   type Boss is new Enemy with
      record
         Multiplier : INTEGER;
      end record;

end EnemyPak2;
```

Body

```
with Text_IO; use Text_IO;
with Ada.Integer_Text_IO;
use Ada.Integer_Text_IO;
package body EnemyPak2 is

   -- Subprograms for Enemy object

PROCEDURE SetDamage(
   AnEnemy : in out Enemy;
   Damage : Integer) IS
BEGIN
   AnEnemy.Damage := Damage;
END;

procedure Attack(
   AnEnemy : Enemy) is
begin
   Put("Attack inflicts ");
   Put(AnEnemy.Damage, 1);
   put_line(" damage units.");
end Attack;

   -- Subprograms for Boss object

procedure SpecialAttack(
   TheBoss : Boss;
   anyEnemy : Enemy'class) is
begin
   Put("Special attack inflicts
");
   Put(AnyEnemy.Damage *
      TheBoss.Multiplier, 1);
   put_line(" damage units.");
end SpecialAttack;

PROCEDURE SetMultiplier(
   TheBoss : in out Boss;
   Multiplier : Integer) IS
BEGIN
   TheBoss.Multiplier :=
      Multiplier;
END;

procedure DisplayDamage(
   Any_Enemy : Enemy'Class) is
begin
   new_line;
   Put("This enemy has damage
capacity of ");
   Put(Any_Enemy.damage, 1);
   Put_line(" units.");
end;

end EnemyPak2;
```

This package shows several important new concepts. First, Enemy is marked Tagged, which means it can be extended by addition of more data and subprograms. In this example, Boss is an extension of Enemy and adds Multiplier as addition member data and SpecialAttack as an additional subprogram. This means Boss inherits Enemy's Attack subprogram, and also gets its own copy of the member data Damage, also inherited from Enemy, in addition to its own member data Multiplier and subprogram SpecialAttack. Enemy is called the Base Class (or superclass) and I derive the Boss class from Enemy when I define it, so Boss is called the derived class (or subclass). In this case, Boss inherits and can directly access Damage and Attack. It's as if I defined both Damage and Attack in Boss.

Because the member data is marked private, I can only set it or read it using member functions setDamage, DisplayDamage, etc. Where I want a function to access data from both the base class and the derived class, I mark the parameter as class-wide.

The main procedure Enemies:

```
with EnemyPak2; use EnemyPak2;
procedure enemies is

   ThisEnemy  : Enemy;
   ThisBoss : Boss;

BEGIN

   SetDamage(ThisEnemy, 10);
   SetMultiplier(ThisBoss, 3);
   SetDamage(ThisBoss, 15);

   DisplayDamage(ThisEnemy);
   Attack(ThisEnemy);

   DisplayDamage(ThisBoss);
   Attack(ThisBoss);
   SpecialAttack(ThisBoss,ThisEnemy);

end Enemies;
```

The program first makes instances of Enemy and Boss, then sets ThisEnemy's member data to 10. It also sets the member data for ThisBoss's member data, Multiplier to 3 and Damage to 15. Note that ThisBoss has inherited member data Damage from Enemy.

It then exercises the member functions. ThisEnemy has only Attack, while ThisBoss has both Attack and SpecialAttack, as it inherits Attack from Enemy.

DisplayDamage is a class-wide procedure so I call it with both ThisEnemy and ThisBoss to show the class-wide action at work. SpecialAttack has its second argument marked as class-wide, so it can use ThisEnemy's weapon instead of its own.

14.2 Controlling Access under Inheritance

When you derive one class from another, you can control how much access the derived class has to the base class's members. For the same reasons that you want to provide only as much access as is necessary to a class's members to the rest of your program, you want to provide only as much access as is necessary to a class's members to a derived class. Not coincidentally, you use the same access modifiers that you've seen before— private and limited private. In this example, we have defined both Enemy and Boss as private, so both can only be accessed via member functions, as is the usual practice.

14.3 Polymorphism

One of the pillars of OOP is polymorphism, which means that a member function will produce different results depending on the type of object for which it is being called. For example, suppose you have a group of bad guys that your hero is facing, and the group is made of objects of different types that are related through inheritance, such as enemies and bosses. Through the magic of polymorphism, you could call the same member function for each bad guy in the group, say to attack the player, and the type of each object would determine the exact results. The call for the enemy objects could produce one result, such as a weak attack, while the call for bosses could produce a different result, such as a powerful attack. This might sound a lot like overriding, but polymorphism is different because the effect of the function call is dynamic and is determined at run time, depending on the object type. But the best way to understand this isn’t through theoretical discussion; it is through a concrete example.

In the bad guy programs above, we see that Boss is actually a special kind of Enemy, so a Boss object is an Enemy object, too. The procedure DisplayDamage(Any_Enemy : Enemy'Class); can be called with argument of either Enemy or Boss and produces the appropriate result.

In Ada, polymorphism is called class-wide programming and dynamic dispatching, which are different words for the same, single concept, which is to allow a client to execute different operations of the same name depending on which type in the class is being accessed. As you can see with DisplayDamage, the procedure is made class-wide with the ‘class attribute.

Most often, polymorphic actions use access types to allow operations to access different versions in the class. A client subprogram can also decide which operation to call by using the membership test IN:

```
  if This IN Leaf then ... end if;
```

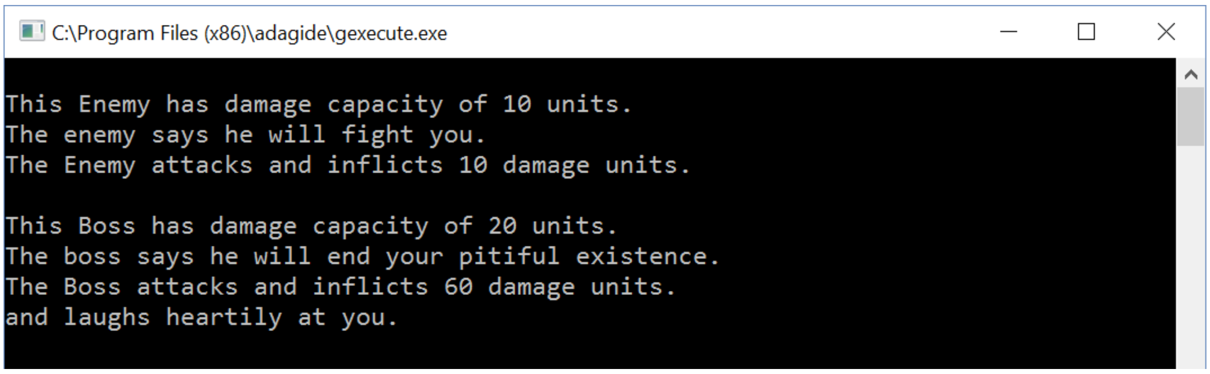
Here we are not sure which derived type variable This belongs to and we want to be sure it belongs to Leaf before taking some action.

14.4 Overriding Base Class Member Functions

Sometimes you will want a derived class to behave differently from the base class. You’re not stuck with every base class member function you inherit in a derived class as is. You have options that allow you to customize how those inherited member functions work in your derived class. You can override them by giving them new definitions in your derived class. You can also explicitly call a base class member function from any member function of your derived class.

14.5 Introducing the Overriding Boss Program

The Overriding Boss program demonstrates calling and overriding base class member functions in a derived class. The program creates an enemy that taunts the player and then attacks him. Next, the program creates a boss from a derived class. The boss also taunts the player and attacks him, but the interesting thing is that the inherited behaviours of taunting and attacking are changed for the boss (who is a bit cockier than the enemy). These changes are accomplished through function overriding and calling a base class member function. Here are the results of running the program:



```
This Enemy has damage capacity of 10 units.
The enemy says he will fight you.
The Enemy attacks and inflicts 10 damage units.

This Boss has damage capacity of 20 units.
The boss says he will end your pitiful existence.
The Boss attacks and inflicts 60 damage units.
and laughs heartily at you.
```

Spec

```
package OvrBossPak is

   type Enemy is tagged private;

   FUNCTION Make(SetDamage : Integer)
      RETURN Enemy;
   PROCEDURE Attack(AnEnemy :
      Enemy'Class;
      multiplier : integer := 1);
-- Attack is a class-wide operation
-- and will accept Enemy or Boss
   PROCEDURE Taunt(AnEnemy : Enemy);

   TYPE Boss IS NEW Enemy WITH
     PRIVATE;

   OVERRIDING FUNCTION Make(
      SetDamage : Integer)
   RETURN Boss;

   Function Make(
      SetDamage : Integer;
      SetMultiplier : Integer)
   RETURN Boss;

   procedure Attack(TheBoss : Boss;
     anyEnemy : Enemy'class);
   overriding procedure Taunt(
     TheBoss : Boss);

-- Display damage is also class-wide
   procedure DisplayDamage(
      Any_Enemy : Enemy'Class);

private

   type Enemy is tagged
      record
         Damage : INTEGER;
      end record;

   type Boss is new Enemy with
      record
         Multiplier : INTEGER;
      end record;

end OvrBossPak;
```

Body

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

package body OvrBossPak is

FUNCTION Make (SetDamage : Integer)
   RETURN Enemy IS
BEGIN
   RETURN (Damage => SetDamage);
end Make;

PROCEDURE Attack(AnEnemy :
   Enemy'Class;
   multiplier : integer := 1) is
begin
  IF AnEnemy IN Enemy THEN
    Put("The Enemy attacks and
inflicts ");
  END IF;
  IF AnEnemy IN Boss THEN
    Put("The Boss attacks and inflicts
");
   END IF;
   Put(AnEnemy.Damage * Multiplier,
     1);
   put_line(" damage units.");
end Attack;

procedure Taunt(AnEnemy : Enemy) is
begin
   Put_Line("The enemy says he will
fight you.");
end Taunt;

   -- Subprograms for the Boss object

  FUNCTION Make (SetDamage : Integer)
     RETURN Boss IS
  BEGIN
     RETURN (SetDamage, 1);
  END Make;

FUNCTION Make (SetDamage : Integer;
     SetMultiplier : Integer)
     RETURN Boss IS
  BEGIN
     RETURN (Damage => SetDamage,
        Multiplier => SetMultiplier);
  END Make;
```

```
PROCEDURE Taunt(TheBoss : Boss) is
BEGIN
   Put_Line("The boss says he will end
your pitiful existence.");
end Taunt;

procedure Attack(TheBoss : Boss;
anyEnemy : Enemy'class) is
begin
   Attack(TheBoss,
      TheBoss.Multiplier);
   Put_Line("and laughs heartily at
you.");
end Attack;

procedure DisplayDamage(
   Any_Enemy : Enemy'Class) is
begin
   New_Line;
   If Any_Enemy in Enemy then
      Put("This Enemy "); END IF;
   IF Any_Enemy IN Boss THEN
      Put("This Boss "); END IF;
   Put( "has damage capacity of ");
   Put(Any_Enemy.damage, 1);
   Put_line(" units.");
END;

end OvrBossPak;
```

Main Procedure

```
with OvrBossPak; use OvrBossPak;

procedure OvrBoss is

   ThisEnemy : Enemy := Make(10);
   ThisBoss : Boss := Make(20, 3);

BEGIN
   DisplayDamage(ThisEnemy);
   Taunt(ThisEnemy);
   Attack(ThisEnemy);

   DisplayDamage(ThisBoss);
   Taunt(ThisBoss);
   Attack(ThisBoss, ThisEnemy);

end OvrBoss;
```

This example is similar to the previous one, with a few changes to demonstrate function overriding. This is accomplished by giving both Enemy and Boss their own versions of Attack and Taunt. Since Boss's Attack has different arguments from Enemy's Attack, Boss Attack does not override Enemy Attack, since the different arguments give them different signatures, so they are recognised as different procedures. Boss Taunt, however, does override its parent, Enemy Taunt, since both arguments are of the same base type. Ada2012 here advocates adding the keyword "overriding". Although this is optional, it avoids a common programming error where the name of the subprogram being overridden is misspelled. Since the parent subprogram and the new subprogram would then have different names, the compiler assumes a new member subprogram is being specified so it would not override the parent. A programmer also has the option of ensuring the new operation does not override the parent by specifying Not Overriding.

In this example we also create constructors for Enemy and Boss which allows us to allocate values to their data members when their instances are created. Given that Ada allows build-in-place aggregates for types, we can return such aggregates from functions. After all, interesting types are usually private, and this provides a neat way for clients to create and initialize objects.

The constructors can have any names, and here I choose to call them Make. There are three versions, Enemy Make, an overridden Enemy Make for Boss (since Boss inherits the Enemy Make) and Boss's own Make, which takes two arguments, Damage and Multiplier. The constructors work because they are public (visible) functions of the Enemy class, so have access to the private data members.

14.6 Initialization and Finalization

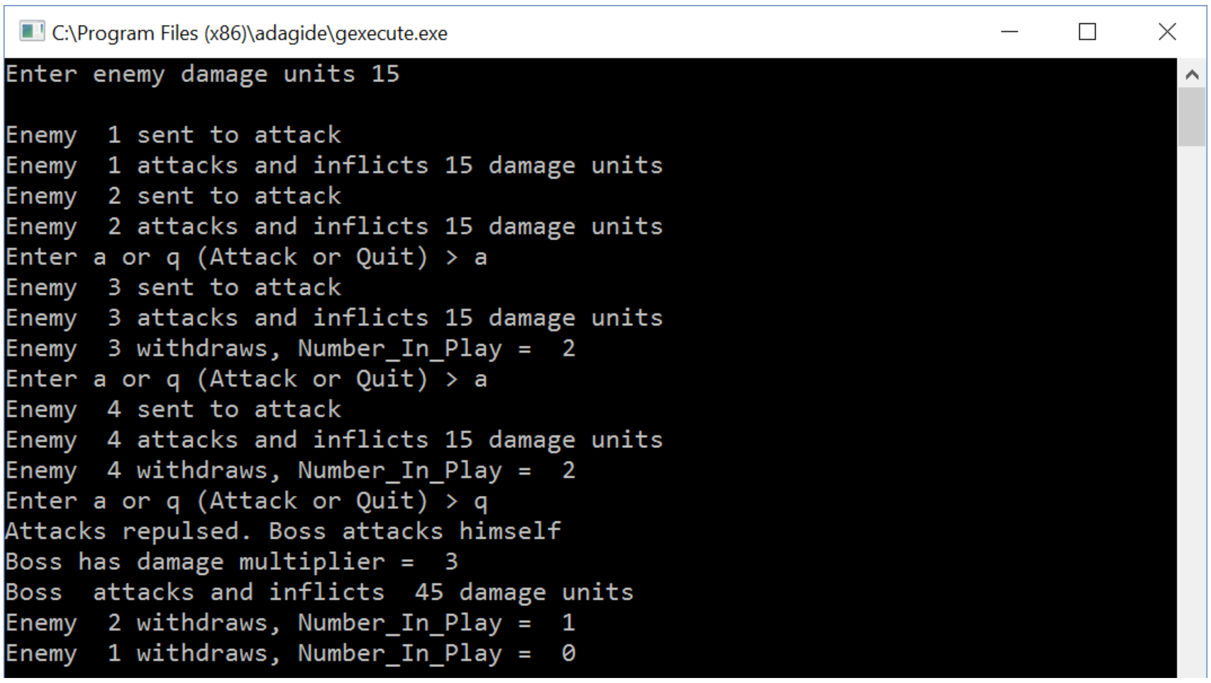
When a new instance of an object is created, it is automatically built by the Ada environment, including allocation of memory from the pool. When it ends and goes out of scope, all resources allocated to it are automatically released, including any memory allocated from the pool. The Ada system is efficient in taking care of its resources, so there is generally no need to be concerned about initializing or terminating an object. If we wish to deallocate an object when it terminates, the unchecked_deallocation procedure you used in the game lobby program is sufficient. On occasion, however, it is useful to automatically execute an initialization procedure when an object is created and a finalization procedure when it terminates. The package Ada.Finalization provides the means to do this. It defines a type, Controlled, which is a special kind of tagged type that gives the programmer this extra control. For example, one can write:

```
type My_New_Type is new Ada.Finalization.Controlled;
```

My_New_Type then inherits three operations called Initialize, Adjust and Finalize. Initialize is automatically called when a controlled object is created. Adjust is automatically called as the last step of an assignment to a controlled object. Finalize is automatically called when a controlled object is about to go out of existence. The default (inherited) versions of these procedures do nothing, but the programmer can override any of them to perform some desired action.

The example will show the use of the package. It extends the Enemy and Boss class to keep track of how many enemies are created. The Boss first send out two Enemies, then can send out additional enemies. Finally, he becomes frustrated, quits sending our Enemies and attacks himself. I have kept the basic structure of the Enemy and Boss class, but now allow for the creation of multiple enemies with automatic Initialization and Finalization. (Note for C++ programmers: these would be called Constructors and Destructors).

Here is the result of running the program:



```
Enter enemy damage units 15

Enemy  1 sent to attack
Enemy  1 attacks and inflicts 15 damage units
Enemy  2 sent to attack
Enemy  2 attacks and inflicts 15 damage units
Enter a or q (Attack or Quit) > a
Enemy  3 sent to attack
Enemy  3 attacks and inflicts 15 damage units
Enemy  3 withdraws, Number_In_Play =  2
Enter a or q (Attack or Quit) > a
Enemy  4 sent to attack
Enemy  4 attacks and inflicts 15 damage units
Enemy  4 withdraws, Number_In_Play =  2
Enter a or q (Attack or Quit) > q
Attacks repulsed. Boss attacks himself
Boss has damage multiplier =  3
Boss  attacks and inflicts  45 damage units
Enemy  2 withdraws, Number_In_Play =  1
Enemy  1 withdraws, Number_In_Play =  0
```

You will notice that Enemies 1 and 2 terminate at the end of the program, while 3 and 4 withdraw and terminate immediately after inflicting their damage. I will explain this below.

Here is the code:

Spec

```
with Ada.Finalization;
package BosswInitPak is
  type Enemy is new
    Ada.Finalization.Controlled
      with private;
```

Enemy is derived from the base type in Ada.Finalization package and has a private section.

```
  procedure Reset_State(E : in out
      Enemy; damage : in integer);
```

Define two member subprograms for Enemy. Reset_State just sets Damage data member

```
  procedure Attack(E  : in Enemy);
```

Attack launches attack with Damage

```
  type Boss is new Enemy
        with private;
```

Now derive Boss from Enemy. It will inherit Enemy's member subprograms and data member Damage

```
  procedure Attack(B : in Boss);
```

Boss gets his own version of Attack.

```
private
  type Enemy is new
    Ada.Finalization.Controlled
       With record
      Number : Natural := 0;
      damage : integer;
    end record;
```

Here is the data record for Enemy

Number will track the number of Enemies sent out.

```
    overriding procedure
     Initialize(B : in out Enemy);
    overriding procedure
     Finalize  (B : in out Enemy);
```

Ensure the Initialize and Finalize procedures are overridden

```
  type Boss is new Enemy
    with record
      Multiplier : integer := 3;
    end record;
```

Here is the data record for Boss. He inherits Damage and adds the new record component Multiplier

```
    overriding procedure
      Initialize(B : in out Boss);
    overriding procedure
      Finalize (B : in out Boss)
       is null;

end BossWInitPak;
```

And make sure Boss get his own Initialize and Finalize procedures. We don't need to do anything when we finalize Boss, so make it NULL (i.e., does nothing).

Body

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

package body BossWInitPak is
  Next_Number : Natural := 0;
  Number_In_Play : Natural := 0;
```

Count the number of Enemies sent out and how many are in play with these two global variables. Enemy Initialize increments both Next_Number and Number_In_Play, Finalize decrements Number_In_Play

```
  procedure Reset_State(E : in out
    Enemy; damage : in integer) is
  begin
    E.Damage := Damage;
  end Reset_State;
---------------------------------
```

Reset_State just sets Enemy data member Damage

```
  procedure Attack(E : in Enemy)
    is
  begin
    Put_line("Enemy "
```

Output damage message for Enemy

```
        & Integer'Image(E.Number)
        & " attacks and inflicts"
        & Integer'Image(E.Damage)
        & " damage units");
  end Attack;
---------------------------------
  procedure Initialize(B : in out
    Enemy) is
  begin
    Next_Number := Next_Number +1;
    Number_In_Play :=
        Number_In_Play + 1;
    B.Number := Next_Number;
    Put_Line("Enemy "
      & Integer'Image(B.Number)
      & " sent to attack.");
  end Initialize;
```

Initialize executes automatically whenever an Enemy instance is created.

It increments the number of Enemies sent out plus the number in play. Save Next_Number in the Enemy data member Number so this Enemy will remember and display which number it is.

```
---------------------------------
  procedure Finalize(B : in out
    Enemy) is
  begin
    Number_In_Play :=
       Number_In_Play - 1;
    Put_Line("Enemy "
       & Integer'Image(B.Number)
       & " withdraws"
       & ", Number_In_Play = "
       & Integer'Image
            (Number_In_Play));
  END Finalize;
```

Finalize decrements Number_In_Play and then displays which Enemy has withdrawn and how many are left in play.

```
---------------------------------
  procedure Attack(B  : in Boss)
    is
  begin
     Put_Line("Boss "
       & " attacks and inflicts "
       & Integer'Image(B.Damage *
         B.Multiplier)
       & " damage units");
  end Attack;
```

Boss attack inflicts Damage * Multiplier

```
---------------------------------
  procedure Initialize(B : in out
     Boss) is
  begin
     Put_Line(
   "Boss has damage multiplier = "
   & integer'image(B.multiplier));
  end Initialize;

END bosswinitpak;
```

Boss initialize just reports Boss's damage Multiplier

Main procedure

```
with bosswinitpak,
Ada.Text_IO, Ada.Integer_Text_IO;
Use  bosswinitpak,
Ada.Text_IO, Ada.Integer_Text_IO;
```

"With" and "Use" all the resources you need, including of course the package for the Enemy and Boss class.

```
procedure BossWInit is
   Ch     : Character;
   damage : integer;
```

We need this for dynamic launching of Enemies

```
   type EnemyRef1_Type is access
     all Enemy;
    EnemyRef1 : array (1..2) of
       EnemyRef1_Type;
```

Create two Enemies in the main Procedure. They will not terminate until the main procedure does.

```
----------------------------------
  Procedure
    Attack_With_Local_Objects
       is
    type EnemyRef2_Type is access
       all Enemy;
    EnemyRef2 : EnemyRef2_Type;
```

Here we will launch additional Enemies within this procedure, so they will go out of scope and therefore terminate when this procedure terminates

```
  begin
    EnemyRef2 := new Enemy;
    Reset_State(EnemyRef2.all,
      damage);
    Attack(EnemyRef2.all);
  end Attack_With_Local_Objects;
```

Make a new Enemy
Set its Damage member data
Launch an Attack and then end, so terminating this procedure and the Enemy it launched. Boss finalization runs automatically, decrements Number_In_Play and displays it.

```
Begin -- main procedure
  Put("Enter enemy damage units ");
  Get(Damage);
  New_Line;
```

Read in a variable Damage to be assigned to each Enemy and Boss.

```
  for I in 1..2 loop
    EnemyRef1(I) := new Enemy;
    Reset_State(EnemyRef1(I).all,
      damage);
    Attack(EnemyRef1(I).all);
  end loop;
```

First send out 2 Enemies by creating new instances. I put them in an array for convenience (and to show it can be done). Then set the Damage member function and launch an Attack. These two Enemies stay in scope until the end of the main procedure.

```
  loop
     Put("Enter a or q (Attack or
Quit) > ");
    Get(Ch);
    exit when Ch = 'q';
    if Ch = 'a' then
      Attack_With_Local_Objects;
    else
      Put_Line("Invalid input");
    end if;
   END LOOP;
```

Loop to send out more enemies. Ask if another is wanted and if a (for another), launch it via procedure Attack_With_Local_Objects, so that the new enemy will terminate at the end of that procedure.

```
   Put_Line("Attacks repulsed. Boss
attacks himself");
   BEGIN
      DECLARE
         type BossRef_Type is access
           all Boss;
      BossRef : BossRef_Type;
```

If user enters q (for Quit), create an instance of Boss to launch a stronger Attack.

```
   BEGIN
      BossRef := new Boss;
      Reset_State(BossRef.all,
         damage);
      Attack(BossRef.all);
    end;
  end;

END BossWInit;
```

Create Boss instance and run auto-Initialization.
Then set Damage member data item

and launch Boss Attack
Boss Attack terminates here
Main procedure and first 2 Enemies terminate here

This program illustrates several key points regarding the scope of instances, the use of Initialization and Finalization and the use of inheritance, where here the program inherits from Ada.Finalization so that Enemy is derived from its base type Controlled and Boss is derived in turn from Enemy. Note that since the Ada system takes care of the basic operations of initializing and finalizing instances, the package Ada.Finalization base procedures do nothing but provide a framework for precise control over allocation, duplication, and deallocation of data storage. These procedures are expected to be overridden by user-defined procedures for the controlled types.

These sorts of types, which provide a platform for the user where the type serves as the foundation for a class of types with certain common properties but without allowing objects of the original type to be declared, are call Abstract Data Types. The user is expected to extend them in order to make them useful. The Ada library packages make heavy use of Abstract types.

14.7 Abstract Data Types

As an example of an Abstract Data Type, suppose you have a game with a bunch of types of creatures running around in it. Although you have a wide variety of creatures, suppose they all have two things in common: they have a health value and they can offer a greeting. So, you could define a class, Creature, as a base from which to derive other classes, such as Pixie, Dragon, Orc, and so on. Although Creature is helpful, it doesn't really make sense to instantiate a Creature object. It would be great if there were a way to indicate that Creature is a base class only, and not meant for instantiating objects. Well, Ada lets you define a kind of class just like this, called an abstract class.

The keyword Abstract is used in the declaration of a type or subprogram that has no instances of its own, but whose concrete descendants do have instances. Here, I will define an abstract type Creature then derive 3 types from it – Orc, Pixie and Dragon. Dragon will also extend the basic type. The code uses inheritance and polymorphism to customise the behaviour of the creatures. I also give each creature its own constructor to set its member data. Here is the code, with commentary

```
Spec                                      Comments

WITH Ada.Strings.Unbounded;
use ada.Strings.Unbounded;

package AbstractCreaturesPak is           -- Begin with a basic abstract
                                          --  Creature type.
 type Creature is abstract tagged
private;                                  -- Then build a type from it

   TYPE Orc IS NEW Creature WITH          -- and start building an instance, Orc
PRIVATE;

   PROCEDURE Greet(AnyCreature :          -- Greet and Health are both class-wide
      Creature'Class);                    -- so can be called with any creature
                                          -- in the class
   Procedure Health(AnyCreature :
      Creature'Class);

   FUNCTION Make(Greeting : String;       -- Initialiser for Orc. Each derived
      health : integer)                   -- creature will need its own.
   RETURN Orc;

   TYPE Pixie IS NEW Creature WITH        -- Now make a pixie along with its
PRIVATE;                                  -- Initialiser.
```

```
   FUNCTION Make(Greeting : String;
      health : integer)
   RETURN Pixie;

   TYPE Dragon IS NEW Creature WITH        -- Now make a Dragon along with its
PRIVATE;                                   -- Initialiser. Its will have an
                                           -- extension to its data members
   FUNCTION Make(Greeting : String;
      health : integer; fire : boolean)
   RETURN Dragon;

private

   type Creature is tagged                 -- Basic member data to be inherited by
      RECORD                               -- all Creatures from acstract Creature
         Greeting : Unbounded_string;
         Health : INTEGER;
      end record;

   type Orc is new Creature with           -- Orc and Pixie do not have additional
      Null record;                         -- member data

   type Pixie is new Creature with
      Null record;

   TYPE Dragon IS NEW Creature WITH        -- Dragon has an additional member data
      RECORD                               -- item
      Fire : Boolean;
   END RECORD;

end AbstractCreaturesPak;
----------------------------
```

Body

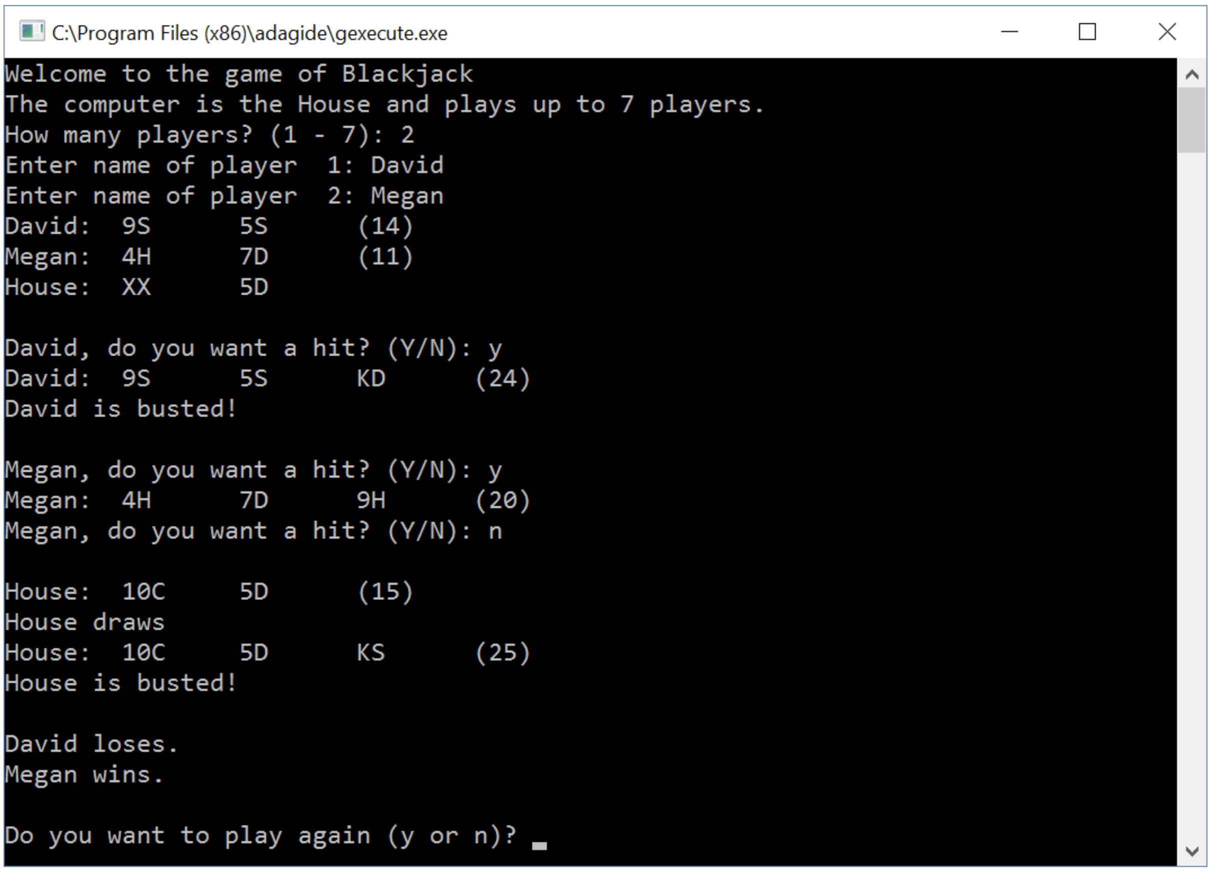
```
                                           -- The body starts as usual
with Text_IO; use Text_IO;
package body AbstractCreaturesPak is

FUNCTION Make(Greeting : String;           -- this builds the initialisers for the
   Health : Integer)                       -- three instances
      return Orc IS
BEGIN                                      -- For Orc and Pixie, return aggregates
     RETURN( To_Unbounded_String(          -- of two items
       Greeting), Health);
end Make;

FUNCTION Make(Greeting : String;
   Health : Integer)
      return Pixie IS
BEGIN
     RETURN( To_Unbounded_String(
        Greeting), Health);
END Make;

FUNCTION Make(Greeting : String;           -- Dragon’s initialiser must return 3
   Health : Integer; Fire : Boolean)       -- items, greeting, health and fire
      return Dragon IS
BEGIN
   RETURN(
To_Unbounded_String(Greeting),
        Health, Fire);
END Make;
                                           -- Greet and Health are both class-wide
PROCEDURE Greet(
     AnyCreature : Creature'Class) is
BEGIN
   New_line;
   Put_line(To_String(
     AnyCreature.greeting));
end Greet;

PROCEDURE Health(
     AnyCreature : Creature'Class) is
begin
   Put_Line("Reports health is "
      & Integer'Image(                      -- Can display Health of all creatures
           AnyCreature.Health)
      & "%");
   IF AnyCreature IN Dragon'Class THEN      -- But Dragon has an additional data
      IF Dragon(AnyCreature).Fire then      -- member, Fire, only valid for it.
        Put_Line("  and is breathing        -- Check if this call is for a Dragon,
fire");                                     -- then type-cast AnyCreature to Dragon
      ELSE                                  -- (required by Ada's strong typing
        Put_line("  but her fire is         -- rules) before displaying a massage
out");                                      -- dependent on the value of Fire
      END IF;
   END IF;
end Health;

end AbstractCreaturesPak;

---------------------------
```

Main procedure

```
WITH AbstractCreaturesPak;
use AbstractCreaturesPak;

procedure AbstractCreatures is              -- Here is the main driver procedure

                                            -- Make and initialise the creatures
   AnOrc : Orc := Make(
     "Orc grunts Hello", 90);
   TYPE OrcRef IS ACCESS Orc;
                                            -- Make a reference to Orc to be able
   Orc2 : OrcRef;                           to
   APixie : Pixie := Make(                  -- create one dynamically
     "Pixie greets you with a twinkle",
      100);
   ADragon : Dragon := Make(
      "Dragon greets you with a hiss of
steam", 100, TRUE);
BEGIN
   Greet(AnOrc);
   Health(AnOrc);                           -- Display the Orc's status

   Orc2 := NEW Orc;
      Orc2.all := Make(                     -- Now make another Orc and initialise
          "Orc2 grunts Hello", 80);         -- it.
   Greet(Orc2.all);
   Health(Orc2.all);                        -- and display the new Orc's status

   Greet(APixie);
   Health(APixie);                          -- Display the Pixie's status

   Greet(ADragon);
   Health(ADragon);                         -- Display the Dragon's status

end AbstractCreatures;
```

15 A larger Object Oriented example – the BlackJack Card Game

This project is a simplified version of the casino card game Blackjack (tacky green felt not included). The game works like this: Players are dealt cards with point values. Each player tries to reach a total of 21 without exceeding that amount. Numbered cards count as their face value. An ace counts as either 1 or 11 (whichever is best for the player), and any jack, queen, or king counts as 10. The computer is the house (the casino) and it competes against one to seven players. At the beginning of the round, all participants (including the house) are dealt two cards. Players can see all of their cards, along with their total. However, one of house's cards is hidden for the time being. Next, each player gets the chance to take one additional card at a time for as long as he likes. If a player's total exceeds 21 (known as busting), the player loses. After all players have had the chance to take additional cards, the house reveals its hidden card. The house must then take additional cards as long as its total is 16 or less. If the house busts, all players who have not busted win. Otherwise, each remaining player's total is compared to the house's total. If the player's total is greater than the house's, he wins. If the player's total is less than the house's, he loses. If the two totals are the same, the player ties the house (also known as pushing). Here is a round of the game:



```
Welcome to the game of Blackjack
The computer is the House and plays up to 7 players.
How many players? (1 - 7): 2
Enter name of player  1: David
Enter name of player  2: Megan
David:  9S      5S      (14)
Megan:  4H      7D      (11)
House:  XX      5D

David, do you want a hit? (Y/N): y
David:  9S      5S      KD      (24)
David is busted!

Megan, do you want a hit? (Y/N): y
Megan:  4H      7D      9H      (20)
Megan, do you want a hit? (Y/N): n

House:  10C     5D      (15)
House draws
House:  10C     5D      KS      (25)
House is busted!

David loses.
Megan wins.

Do you want to play again (y or n)?
```

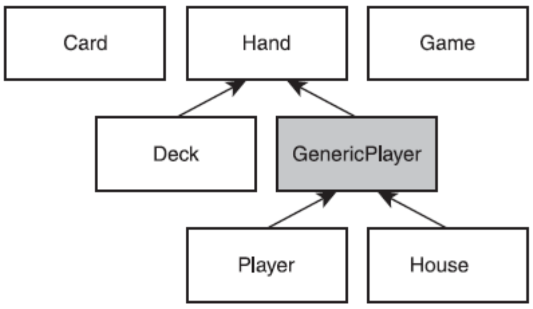
15.1 Designing the Classes

Before you start coding a project with multiple classes, it is helpful to map them out on paper. You might make a list and include a brief description of each class. This table shows my first pass at such a list for the Blackjack game.

Class	Base Class	Description
Vector	Containers	Holds hands, deck, players

Cards	None	Definition of playing cards, ranks, suits
Hands	None	A blackjack hand is a collection of card objects
Deck	None	The deck is a full set of playing cards from which the House deals
GenericPlayer	None	Abstract player from which Player and House are derived
Players	GenericPlayer	Human player with Name and Hand
House	GenericPlayer	The Computer player, deals cards to players
Game	None	Runs the game
BlackJack	None	Main Program

This set of objects could all be created as separate packages, or as one big package, or they can be further grouped. Hands and Decks have much in common so I chose to group them together as one package, and also grouped the generic player, the players and the House into one package. There are lots of options here and this is just one way to organise the code.



Also, to keep things simple, all member functions and all data members will be public and each derived class will inherit all of its base class members.

In addition to describing your classes in words, it helps to draw a family tree of sorts to visualize how your classes are related. That's what I did in this diagram.

Next, it's a good idea to get more specific. Ask yourself about the classes. What exactly will they represent? What will they be able to do? How will they work with the other classes?

I see Card objects as simulating real-life cards. Each card has a Rank (A, 2 … K) (for Ace, 1, … King) and a Suit (C, D, H, S) (for Clubs, Diamonds, etc.). These can be set up as enumeration types and a card may thus be represented as a record containing its Rank, Suit and a Boolean to indicate if it is face up, because the House gets dealt a card that is face down. The Deck will be a vector of cards, as will each Player's Hand. When the house deals a card, it gets moved from the deck to a hand, not copied. That means that when a card moves from the Deck to a Player's Hand, the card vector element will be copied from the Deck, added to the Hand, then the copy in the Deck will be deleted.

I see players (the human players and the computer) as Blackjack hands with names. That's why each player and the House has a Hand as a data member. Since the House and the Players have much in common, I define GenericPlayer to have the functionality that Player and House share, as opposed to duplicating this functionality in both classes. The Deck is a vector of Cards and will be a data member of the Game object. During the Game, cards are dealt from the Deck to the human players and the computer-controlled House in the same way. This means that the function to deal cards is polymorphic and will work with either a Player or a House object.

To really flesh things out, you can list the data members and member functions that you think the classes will have, along with a brief description of each. That's what I do next, using the Ada specifications to list the definitions for each class. Since this is a larger program that you have seen so far, I will adopt the convention of labelling each member data item as m_item, so they are clearly recognisable as data members.

Now I need to think about organising the classes in packages. One option is to make every class a separate package, but that way, in any reasonably-sized project, you end up with too many packages. So I'll will bundle similar or related classes into packages. Cards, Hands and the Deck are all card-related, so I'll put them in one package, which I'll call Cards, and the House and Players are all player related, so I'll put them together into a Players package. The Game package inherits from both, so I'll keep it in its own package.

15.2 The Cards Package

Here is package Cards, which specifies the Card, Hand and Deck classes. Each card will have 3 data members Rank (Ace,2,3…King) and a Suit (C,D,H,S for Clubs, Diamonds, Hearts and Spades) plus a Boolean for IsfaceUp.

To make a Hand and a Deck I just need a vector of cards, so they are basically the same thing. Each Player and the House has a Hand, a Deck will be part of the Game object and will need some additional functions to create and shuffle the Deck. Both the Hand and the Deck can use the same container, namely a vector of Cards. A Hand and a Deck do not need additional data members, just a set of functions to operate on the Hand and Deck vectors and these are provided by the Vector container package.

So here is the Cards package, including the functions for Hand and Deck:

```ada
WITH Ada.Containers.Indefinite_Vectors; USE Ada.Containers;

PACKAGE Cards IS

-- Can't make this (A, 2 .. etc as components of an enumeration type
-- must all be literals, so add the v to make 2, 3 etc. into literals
type Rank is (vA, v2, v3, v4, v5, v6, v7, v8, v9, v10, vJ, vQ, vK);

TYPE Suit IS (C, D, H, S);

TYPE Card IS record
      m_Rank : Rank;
      m_Suit : suit;
      m_IsFaceUp : Boolean;
   END RECORD;

PACKAGE CardVectors IS NEW Indefinite_Vectors (Natural, Card);
use CardVectors;   -- CardVectors is used for Hands and the Deck

-------- functions for cards

-- Display one Card eg AS for Ace of Spades
Procedure DisplayCard(C : Card);

-- Flip card, face up to face down or vice versa
Procedure Flip(C : in out Card);

-- Calculate the value of a card, A = 1, Face cards = 10
FUNCTION GetValue(C : Card) RETURN Natural;

-------- functions for hands

-- Total vaue for a Hand (Player or House)
FUNCTION GetTotal(H : Vector) RETURN Natural;
```

```
------ Functions for Deck (uses same vector container as Hands)

--create a standard deck of 52 cards
PROCEDURE Populate(D : IN OUT Vector);

-- shuffle deck of cards
procedure Shuffle(D : in out Vector);

-- deal one card to a hand
procedure Deal(D : in out vector;  H : in out vector);

End Cards;
```

15.3 The Players Package

The Players package specifies a generic Player, then derives the House and the actual Player type. I can write single subprograms for IsBusted and Bust, but Player and House require different bodies for IsHitting because the rules differ for when the House and the Players can receive a hit (i.e. an additional card). So House and Player each get specific subprograms of their own and so are most simply created as separate types that inherit from genericPlayer. The one function that needed some thought is AdditionalCards, since both House and Player can draw cards, so this is a function of Cards and Players. Initially I put it in the Cards package, but then moved to the Player package because it is a generic function for both House and Player. The other question I needed to address was whether to make the Players an array, since there are up to 7 players, or another vector. Since one of the data members for type Player will be the player’s name, and its variable in length, it seemed easiest to use an indefinite vector for this.

So here is the spec for package Players. It defines a generic player from which it derives the House and the Player instances.

```
WITH Ada.Strings.Unbounded; USE Ada.Strings.Unbounded;
WITH Ada.Containers.Indefinite_Vectors; USE Ada.Containers;
WITH cards;

PACKAGE Players IS

   TYPE GenericPlayer IS ABSTRACT TAGGED;

-- indicates whether or not generic player wants to keep hitting
-- The house and a Player have different rules, so have different code
   FUNCTION IsHitting (P : GenericPlayer) RETURN Boolean IS ABSTRACT;

-- returns whether generic player has busted - has a total greater than 21
   FUNCTION IsBusted (P : GenericPlayer) RETURN Boolean;

-- announces that the generic player has bust
   PROCEDURE Bust (P : GenericPlayer);

-- Display a generic Player's Hand and its total
   PROCEDURE DisplayHand(Gen : GenericPlayer'class);

-- give additional cards to a generic player
   PROCEDURE AdditionalCards(D : IN OUT Cards.CardVectors.Vector;
                              Gen : IN OUT GenericPlayer'Class);

   TYPE GenericPlayer IS ABSTRACT TAGGED
      RECORD
         M_Name : Unbounded_String; -- each player has a name
         M_Hand : cards.CardVectors.Vector;    -- and a hand
      END RECORD;

   ---------- House has its own code but no new data members

   TYPE House IS NEW GenericPlayer with NULL record;

-- indicates whether house is hitting - will always hit on 16 or less
   FUNCTION IsHitting(H: House) Return Boolean;

-- flips over first card
   Procedure FlipFirstCard(H : in out House);

   ----------- Player has its own code, but also no new data members

   TYPE Player IS NEW GenericPlayer with NULL record;

-- returns whether or not the player wants another hit
   FUNCTION IsHitting(P : Player) RETURN Boolean;

-- announces that the player wins
   procedure Win(P : Player);

-- announces that the player loses
   procedure Lose(P: Player);

-- announces that the player pushes
   PROCEDURE Push(P: Player);

-- Now make a vector to hold a list of players
   Package PlayerVectors is new Indefinite_Vectors(Natural, Player);

END Players;
```

15.4 Designing The Game Package

The Game Package builds a vector of Players, which is a different type of vector from Hands. Its data members are the list of Players, the House and the Deck. Its functions are to set up a Game, initialise it by setting up a list of Players, building and shuffling a Deck and then playing the Game according to the game logic.

Having decided on the data we need, we can plan the Game.

I will start by mapping out the basic flow of one round of the game. I wrote some pseudocode for the Game class Play member procedure. Here's what I came up with:

```
Deal players and the house two initial cards
Hide the house's first card
Display players' and house's hands
Deal additional cards to players
Reveal house's first card
```

```
Deal additional cards to house
If house is busted
    Everyone who is not busted wins
Otherwise
    For each player
        If player isn't busted
            If player's total is greater than the house's total
                Player wins
            Otherwise if player's total is less than house's total
                Player loses
            Otherwise
                Player pushes
Remove everyone's cards
```

At this point you know a lot about the Blackjack program and you haven't even seen any code (apart from the specs)! But that's a good thing. Planning is as important as coding (if not more so). Because I've spent so much time describing the classes, I won't describe every part of the code. I'll just point out significant or new ideas.

The blackjack program contains seven classes. In Ada programming, it's common to break up programs like this into multiple files, with classes grouped into packages, as I have done in the above specs. This makes the packages easier to work on, rather than having one big package.

I showed the specs from the bottom up (lowest class first). Now let's describe the code from the top down, starting with the main program, BlackJack.

With the above set of packages, all BlackJack has to do is display a startup message, initialise the game, call Play and when a game round finishes, ask if the user wants to play another round:

```
WITH Text_IO; USE Text_IO;
with Game;

PROCEDURE Blackjack IS
   C: Character;
   G: Game.GameT;  -- make an instance of the Game
BEGIN
   Put_Line("                Welcome to the game of Blackjack");
   Put_line("       The computer is the House and plays up to 7 players.");
   Game.InitGame(G);  -- Initialise the Game (House, Players and Deck)
   LOOP
      Game.Play(G);   -- Play a round of the game
      LOOP
         new_line;
         Put("Do you want to play another round (y or n)? ");
         Get(C); Skip_Line;
         exit when C IN 'y' | 'Y' | 'n' | 'N';  -- Ada2012 feature
      END LOOP;
      exit WHEN C IN 'n' | 'N';  -- end the game
   END LOOP;                     -- or play another round with the same
END;                             -- set of players and deck of cards.
```

The main program, as is often the case in Ada, is very simple. The game logic is all captured in the Play procedure in the Game package. So let's look at the Game package body to see what it does.

15.5 Implementing the Game Package

I will deliberately not make the Cards and Hands packages directly visible with “Use”. This forces me to name the package each time I use a subprogram in it and this helps when reading the code, although it means using longer fully qualified names.

Game Package body:

```
WITH Cards, Hands;   -- usual preamble, listing packages “with”ed
WITH Text_Io; USE Text_Io;
WITH Ada.Integer_Text_IO; USE Ada.Integer_Text_IO;
WITH Ada.Strings.Unbounded; USE Ada.Strings.Unbounded;
WITH Ada.Text_IO.Unbounded_IO; use Ada.Text_IO.Unbounded_IO;
WITH Ada.Strings; USE Ada.Strings;

PACKAGE BODY Game IS

   HT : CONSTANT Character := Character'Val(9);  -- Horizontal Tab

   PROCEDURE InitGame (G : IN OUT GameT) IS
   BEGIN
      Hands.Populate(G.M_Deck);  -- build a standard deck of 52 cards
      Hands.Shuffle(G.M_Deck);   -- shuffle the deck
      AddPlayers(G);             -- Get list of players
      G.m_House.m_Name := to_unbounded_string ("House"); -- House’s name!
   END InitGame;

   PROCEDURE Play (G : IN OUT GameT) IS   -- plays a round of blackjack
      PTot, HTot : Integer;
   BEGIN
      -- deal initial 2 cards to everyone and display their hands
      FOR P OF G.M_Players LOOP      -- iterator through list of players
         Hands.Deal(G.M_Deck, P.M_Hand);  -- deal 2 cards to each player
         Hands.Deal(G.M_Deck, P.M_Hand);  -- from deck to player’s hand
         Cards.Display_Hand(P.M_Hand, to_string(P.M_Name)); -- display them
      END LOOP;

      -- now deal 2 cards to the house, from Deck to House’s Hand
      Hands.Deal(G.M_Deck, G.M_House.M_Hand);
      Hands.Deal(G.M_Deck, G.M_House.M_Hand);

      --hide house's first card
      FlipFirstCard(G.M_House);

      Put("House:" & HT);   -- Show House’s cards (first is face down)
      Put("XX" & HT);       -- use horizontal tabs for display formatting
            -- then display House’s 2nd card ie next card after first
      Cards.DisplayCard(Element(next(First(G.m_House.m_Hand))));
      New_Line;

      -- deal additional cards to all players who ask for them
      FOR P OF G.M_Players LOOP
         Hands.AdditionalCards(G.M_Deck, P);
      END LOOP;

      -- reveal house's first card
      FlipFirstCard(G.M_House);

      -- and display House's cards
      New_line;
      Hands.Display_Hand(G.M_House.M_Hand, G.M_House.M_Name);
      IF Hands.GetTotal(G.M_House.M_Hand) > 16 THEN
         Put("House sticks");
      ELSE
         Put("House draws");
      END IF;

      -- deal additional cards to house (if needed)
      Hands.AdditionalCards(G.M_Deck, G.M_House);
      new_line;
      IF (G.M_House.IsBusted) THEN
         -- everyone still playing wins
         FOR P OF G.M_Players LOOP
            IF P.IsBusted THEN
               Lose(P);
            ELSE
               Win(P);
            END IF;
         END LOOP;
      ELSE
         -- compare each player still playing to house
         Htot := Hands.GetTotal(G.M_House.M_Hand);  -- Houses’s total
         FOR P OF G.M_Players LOOP
            IF ( NOT  P.IsBusted) THEN
               PTot := Hands.GetTotal(P.M_Hand);  -- Get Player’s total
               IF PTot > HTot THEN        -- compare it to House’s total
                  Win(P);
               ELSIF PTot < HTot THEN
                  Lose(P);
               ELSE
                  Push(P);
               END IF;
            ELSE
               Lose(P);  -- P is busted, house isn't, so player loses
            END IF;
         END LOOP;
      END IF;

      -- round is complete so clear everyone's hands
      FOR P OF G.M_Players LOOP
         Clear(P.M_Hand);   -- vector function clears player’s hands
      END LOOP;
      Clear(G.M_House.M_Hand);  --Clear House’s hand
   END Play;


   PROCEDURE AddPlayers ( G : IN OUT GameT) IS
      NumPlayers : Integer := 0;
      P          : Player;
   BEGIN
      WHILE (NumPlayers < 1 OR NumPlayers > 7) LOOP
         Put( "How many players? (1 - 7): ");
         Get(NumPlayers);
         Skip_Line;
      END LOOP;

      FOR I IN 1..NumPlayers LOOP
         Put( "Enter name of player " & Integer'Image(I) & ": ");
         Get_Line(P.M_Name);  -- Get player’s name
         Clear(P.M_Hand);     -- Clear player’s hand
         G.M_Players.Append(P);  -- Add player to list of players
      END LOOP;

   END AddPlayers;

END Game;
```

Now let’s look at the code for the players and the House. These subprograms are simple.

```
WITH Text_Io; USE Text_Io;

PACKAGE BODY Players IS

   -- returns whether generic player has busted - has total greater than 21
   FUNCTION IsBusted(P : GenericPlayer) RETURN Boolean IS
   BEGIN
      RETURN (cards.GetTotal(P.M_Hand) > 21);
   END;

   -- announces that the generic player busts
   PROCEDURE Bust(P : GenericPlayer) IS
   BEGIN
      Put_Line(To_String(P.M_Name) & " is busted!");
   END;

   -- give additional cards to a generic player
   PROCEDURE AdditionalCards (
         D : IN OUT cards.CardVectors.vector;
         Gen : IN OUT GenericPlayer'Class) IS  -- either player or house
   BEGIN
      New_Line;
      -- continue to deal a card as long as generic player isn’t busted and
      -- hasn’t got 21 and asks for another hit
      WHILE  (NOT IsBusted(Gen))   -- use AND THEN to specify the order in
              AND THEN GetTotal(Gen.m_Hand) /= 21  -- which the conditions
              AND THEN IsHitting(Gen) LOOP         -- are evaluated
         Deal (D, Gen.M_Hand);     -- wants a card so deal one
         Display_Hand(Gen.m_Hand, to_string(Gen.m_Name));
         IF IsBusted(Gen) THEN
            Bust(Gen);
         END IF;
      END LOOP;
   END;


   ------- Functions for House

   -- indicates whether house is hitting - will always hit on 16 or less
   FUNCTION IsHitting ( H : House )
     RETURN Boolean IS
   BEGIN
      RETURN (cards.GetTotal(H.M_Hand) <= 16); -- just return the condition
   END;

   -- flips over first card
   PROCEDURE FlipFirstCard ( H : IN OUT House) IS
      C : Cards.Card;
   BEGIN
      IF Cards.CardVectors.Is_Empty(H.M_Hand) THEN
         Put_Line("No cards to flip!");
      ELSE
         C := Cards.CardVectors.First_Element(H.M_Hand); --get the card
         Cards.Flip(C);                                  --Flip the card
         Cards.CardVectors.Replace_Element(   -- replace first card with
            H.M_Hand, cards.CardVectors.First(H.M_Hand), C);  -- flipped
      END IF;
   END;

   --------- Functions for Players

-- ask if the player wants another hit, return TRUE if yes
   FUNCTION IsHitting(P : Player) RETURN Boolean IS
      c : character;
   BEGIN
      put(to_string(P.m_Name) & ", do you want a hit? (Y/N): ");
      get(c);
      RETURN (C =  'y' OR C = 'Y');
   END;

   -- announces that the player wins
   PROCEDURE Win(P : Player) IS
   BEGIN
      Put_line(to_string(P.m_Name) & " wins.");
   END;

-- announces that the player loses
   PROCEDURE Lose(P: Player) IS
   BEGIN
      Put_line(to_string(P.m_Name) & " loses.");
   END;

-- announces that the player pushes
   procedure Push(P: Player) is
   BEGIN
      Put_line(to_string(P.m_Name) & " pushes.");
   END;

END Players;
```

15.6 The Cards Package Body

Well, so far, so good. We've got the game package, the Players and the House. Now we need to give them cards, hands and the deck, and these are in the package Cards. Again, the coding for the member functions is simple.

```
WITH Text_Io; USE Text_Io;
WITH Ada.Numerics.Discrete_Random;  -- need this for shuffle

PACKAGE BODY Cards IS

   PROCEDURE DisplayCard(C : Card) IS
      R : Rank;
      HT : constant Character := Character'Val(9);  -- see character set
   BEGIN
      R := C.M_Rank;
      IF R = v10 THEN Put("10");
      -- display second character of enumeration type vA, v2 etc
      ELSE Put(Rank'Image(R)(2));
      end if;
      text_IO.Put(suit'image(C.M_Suit) & HT);  -- then the suit and a tab
   END;

   PROCEDURE Flip(C : IN OUT Card) is
   BEGIN
      C.M_IsFaceUp := NOT C.M_IsFaceUp;
   end;

   FUNCTION GetValue(C : Card) RETURN Natural IS
      value : Natural := 0;
   BEGIN
      -- if a cards is face down, its value is 0
      IF C.M_IsFaceUp THEN
         -- value is number showing on card, 1 for Ace
         -- but enumeration type starts from 0, so add 1
         Value := Rank'Pos(C.M_Rank) + 1;
         -- value is 10 for all face cards
         IF (Value > 10) THEN
            Value := 10;
         END IF;
      END IF;
      RETURN Value;
   END;

------ functions for hands

   FUNCTION GetTotal(H : Vector) RETURN Natural IS
      Total : Natural := 0;
      ContainsAce : Boolean := False;
   BEGIN
      -- if hand is empty, return 0
      IF Is_Empty(H) THEN RETURN 0;
      END IF;
      -- add up card values, treating each ace as 1
      -- but record if the hand contains an ace
      FOR EachCard OF H LOOP
         Total := Total + Cards.GetValue(EachCard);
         IF EachCard.m_Rank = vA THEN
            ContainsAce := True;  -- yes, hand contains an ace
         END IF;
      END LOOP;
      -- if hand contains ace and total is low enough, treat ace as 11
      IF (ContainsAce AND Total <= 11) THEN
      -- add only 10 since we've already added 1 for the ace
         Total := Total + 10;
      END IF;
      return total;
   END;

   ------ Subprograms for Deck

   --create a standard deck of 52 cards
   PROCEDURE Populate ( D : IN OUT Vector) IS
      C : Cards.Card;
   BEGIN
      Clear(D);  -- vector function
      FOR S IN Cards.Suit LOOP
         FOR R IN Cards.Rank LOOP
            C.M_Rank := R;    -- makes a deck containing
            C.M_Suit := S;    -- 52 cards every rank & suit
            C.M_IsFaceUp := True;  -- cards are dealt face up
            Append(D, C);
         END LOOP;
      END LOOP;
   END;

   -- shuffle cards
   PROCEDURE Shuffle ( D : IN OUT Vector) IS
      SUBTYPE Deck_Size IS Integer RANGE 0..51;
      PACKAGE RandomCard IS NEW
         Ada.Numerics.Discrete_Random(Deck_Size);
      USE RandomCard;  -- makes a random number in range 0..51
      G     : Generator;  -- for the random numbers
      Other : Integer;    -- used for a random index
   BEGIN
      Reset(G);
      FOR E IN First_Index(D)..Last_Index(D) LOOP
         Other := Random(G);  -- now swap each card in the deck
         Swap(D, E, Other);   -- with another in the deck at random
      END LOOP;
   END;

   -- deal one card to a hand
   PROCEDURE Deal ( D : IN OUT Vector; H : IN OUT Vector) IS
      C : Cards.Card;
   BEGIN
      IF Is_Empty(D) THEN
         Put_Line("Out of Cards, take a new pack.");
         Populate(D);
         Shuffle(D);
      END IF;
      -- Deal a card from the deck to the hand
      C := First_Element(D);  -- take the first card from the deck
      Delete_First(D);        -- then delete it from the deck
      Append(H, C);           -- and add it to the hand
   END;

End Cards;
```

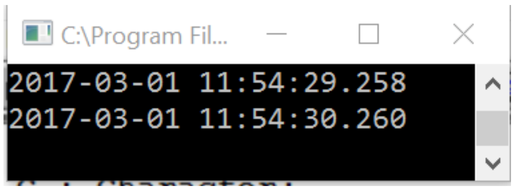
And that's it for this version. Of course the casino game is more interesting because it includes betting, where players can vary their bets and accumulate their winnings, but the basic structure is there so that the program is easy to extend to include betting – just add two member data items to the players package, m_Winnings and m_Bet, then include member functions to place a bet, add it to the player's winnings on a win and deduct it on a loss, and display the player's winnings. You could also extend the Game package to allow players to leave the game and others to join.

The game code is all in a subdirectory, Blackjack, in the examples folder.

I'll make a comment on developing this game. It contains 4 packages along with the main procedure, so needed much thought and testing. I wrote small test programs for all the packages as I developed them, following my earlier advice to code a little and test a lot. This helped a great deal with nuilding the code in an organised way as well as with the debugging.

One of the key features of Ada, as I pointed out in Chapter 1, is that Ada supports a full set of time, date and duration (time period) types as well as a calendar package with time zones and the like. I have already used the delay statement in the Hanoi program to slow down the display, and used the time variable to time the run-up in the Javelin throw example. This is great for games programmers who want to keep track of and manipulate variables of these types. Let's start with a simple example of a small digital clock that ticks once per second.

```
WITH Text_IO, ada.calendar, Ada.calendar.Time_Zones,
ada.calendar.formatting;
USE Text_IO, ada.calendar, Ada.Calendar.Time_Zones,
ada.calendar.formatting;
PROCEDURE Time_drift IS
   Time_Now : Time;
   C : Character;
   avail : boolean;
BEGIN
   LOOP
      time_now := clock;
-- function Image (Date : Time;  -- function from Calendar.Formatting
--      Include_Time_Fraction : Boolean := False;
--      Time_Zone  : Time_Zones.Time_Offset := 0) return String;
      Put(Ada.Calendar.Formatting.Image(Time_Now, False, 8*60));
      Put(".");
      Put_Line(second_duration'image(sub_second(time_now))(4..6));
-- function Sub_Second (Date : Time) return Second_Duration;
      DELAY 1.0;
      Get_Immediate (C, Avail); -- Avail is True is user hits any key
      EXIT WHEN Avail;          -- and trigger Exit
   end loop;
END Time_drift;
```



```
2017-03-01 11:54:29.258
2017-03-01 11:54:30.260
```

If you run it, then shrink the console window, you get output like that shown, which steps by one second at a time. Note the Time Zone offset is set for Sydney Australia winter time. We want the time displayed to step by exactly one second, but as can be seen, there is a small error, from 29.258 to 39.260, an error of 0.002 of a second.

The simple Delay statement in the above code does delay for one second and in most programs, where all that is needed is a short delay, a simple delay statement is enough. But a clock needs more precise timing. A loop of the form shown on the left will take longer to repeat the loop that the period set by the delay statement, due to the execution time taken by the processing, plus the time taken to suspend and then resume the execution of the program, so the delay will accumulate and the clock will slowly lose time and eventually miss a second.

```
LOOP
   <Some Processing>;
   DELAY <Some Duration>;
END LOOP;
```

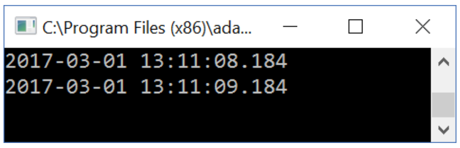
For our clock, we'd like the loop to repeat exactly every second and not suffer from the cumulative delay, which is called timing drift. To achieve this we use the delay until <some time> statement, where the argument is a fixed time rather than a duration. Then we step the fixed time along by a

```
LOOP
   <Some Processing>
   next_time := next_time + Some_period;
   DELAY until next_time;
END LOOP;
```

second at a time and the delay until waits until that specified time for the loop to repeat. So now our loop takes the slightly different form shown. Next_time must be of type Time, from Calendar, while Some_period is the time we want the loop to repeat. For our clock, that is every second. In real time and embedded control systems, timing precision can be critically important. In games, simple delays are usually good enough, unless we are programming a clock or need precise timing. Ada also has a real-time package with more precise timers, but that is a topic for another time.

Here is the modified code and corresponding output.

```
WITH Text_IO, Ada.Calendar, Ada.Calendar.Time_Zones,
Ada.Calendar.Formatting;
USE Text_IO, Ada.Calendar, Ada.Calendar.Time_Zones,
Ada.Calendar.Formatting;
PROCEDURE No_Time_Drift IS
   Next_Time : Time                   := Clock;
   Time_Now  : Time;
   C         : Character;
   Avail     : Boolean;
   TZ        : Time_Zones.Time_Offset := 8 * 60; -- minutes
BEGIN
   LOOP
      Time_Now := Clock;
      -- function Image (Date : Time;
      --    Include_Time_Fraction : Boolean := False;
      --    Time_Zone  : Time_Zones.Time_Offset := 0) return String;
      Put(Image(Time_Now, False, TZ));
      Put(".");
      Put_Line(Second_Duration'Image(Sub_Second(Time_Now))(4..6));
      -- function Sub_Second (Date : Time) return Second_Duration;
      Next_Time := Next_Time + 1.0;
      DELAY UNTIL Next_Time;
      Get_Immediate (C, Avail);
      EXIT WHEN Avail;
   END LOOP;
   New_Line;
   Put_Line("Program  terminated by user hitting a key");
END No_Time_Drift;
```



```
2017-03-01 13:11:08.184
2017-03-01 13:11:09.184
```

As you can see when you run the program, there is now no time drift. Each output is one second apart. In bigger, more complex programs, there might be some variation in the timing each second (called local variation), but it does not accumulate.

Timing drift can cause serious timing errors in some types of games and you need to be aware of this issue so you can avoid it when necessary.

Exercise: extend this program so it starts at the next precise second, rather than starting at some random fraction of a second.

16.1 Revisiting the game loop

Back in chapter 2, I talked about the game loop. In most of our games so far, the game loop has not included delays. But we have also written some code where the game loop includes a delay, and if we want precise timing, we can use a delay until. Here are 3 versions of the game loop:

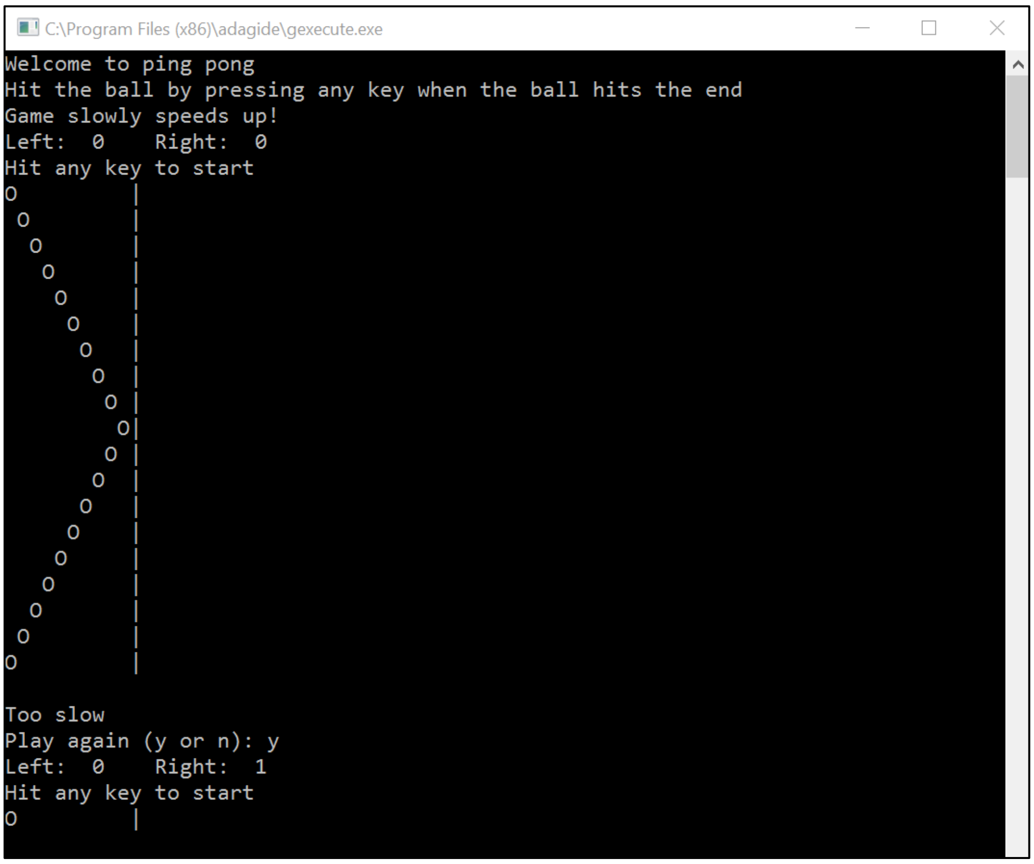
```
Setup
Loop
   Wait for user inputs
   Do game calcs
   Output
End loop
```

```
Setup
Loop
   Any user inputs?
   Do game calcs
   Output
   Delay Period
End loop
```

```
Setup
Loop
   Any user inputs?
   Do game calcs
   Output
   Delay until Next Time
End loop
```

Precise timing may be needed in games with graphics where the graphical output must be generated at a specific frame rate in order to preserve the realism of the graphics. It is also very useful to ensure a game runs at the same rate, irrespective of the speed of the hardware. Since newer machines might be much faster than older ones, if the game loop does not have a precise delay the game loop might run too fast for the game to playable. So a game loop with timing performs a key task: it runs the game at a consistent speed despite differences in the underlying hardware.

16.2 A ping pong game



```
Welcome to ping pong
Hit the ball by pressing any key when the ball hits the end
Game slowly speeds up!
Left:  0    Right:  0
Hit any key to start
O         |
 O        |
  O       |
   O      |
    O     |
     O    |
      O   |
       O  |
        O |
         O|
        O |
       O  |
      O   |
     O    |
    O     |
   O      |
  O       |
 O        |
O         |

Too slow
Play again (y or n): y
Left:  0    Right:  1
Hit any key to start
O         |
```

This game simulates a simplified game of ping pong. It uses simple delay statements for timing and slowly speeds up as the game progresses by reducing the delay in the game loop. Many similar games that test a player's reflexes implement this kind of game loop.

The game can be played by one or two players, since no fixed key is assigned for hitting the ball. The game displays a ball (an ‘O’) moving across the screen and the player(s) must hit it at the end of its travel by pressing any key. If the player hits it, the ball travels in the opposite direction, moving slightly faster, so the game continuously speeds up until a player misses by trying to hit it too soon or too late, and loses the point. The game then asks if the player wishes to play again. The running total score is displayed at the start of each round. A typical output screen is shown above.

The pseudocode for this game is very simple.

```
Set the starting direction to move to the right
Hit a key to start
Loop until a player misses
    Move the ball (a letter ‘O’) across the screen
    At the end of the screen, if the player hits a key while the ball is at the end
        Change direction
        Speed up the game
    Else exit
End Loop
```

Here is the code for the ping pong game:

```
WITH Ada.Text_IO; USE Ada.Text_IO;

PROCEDURE Pingpong IS
   BlankOutput : CONSTANT String  := "          |"; -- set up the ‘table’
   Len         : CONSTANT Integer := BlankOutput'Length - 1;
   Output      :          String  := BlankOutput;
   C           :          Character;
   Avail       :          Boolean;
   InitSpeed   :          Duration := 0.6;
   Speed       :          Duration := InitSpeed;
   Speedup     : CONSTANT Float    := 0.9;
   Left, Right :          Integer  := 0;

   FUNCTION GoodHit RETURN Boolean IS
   BEGIN
      Get_Immediate(C, Avail);  -- check the player didn’t hit too soon
      IF Avail THEN
         New_Line;           -- player already hit - it’s too soon
         Put_Line("Too fast");
         RETURN False;
      END IF;
      DELAY Speed;           -- wait for a hit
      Get_Immediate(C, Avail);  -- should have a hit now
      IF NOT Avail THEN      -- If not, it’s too late
         New_Line;
         Put_Line("Too slow");
         RETURN False;
      END IF;
      RETURN True;              -- Good hit - not too slow and not too fast
   END GoodHit;

BEGIN  -- main program
   Put_Line("Welcome to ping pong, for 1 or 2 players");
   Put_Line("Hit the ball by pressing any letter key when the ball hits the
end");
   Put_Line("Game slowly speeds up!");
   LOOP
      Put_Line("Left: " & Integer'Image(Left)
                        & "    Right: " & Integer'Image(Right));
      Output := BlankOutput;
      Speed := InitSpeed;
      Output(1) := 'O';    -- starts at left of screen
      Put_Line("Hit any key to start");
      Put_Line(Output);    -- show starting position
      Get_Immediate(C);
      LOOP
         FOR I IN 2..Len LOOP
            Output(I) := 'O';    -- move ball across the screen
            Output(I-1) := ' ';  -- left to right
            Put_Line(Output);
            IF I /= Len THEN     -- delay at end is handled in GoodHit
               DELAY Speed;      -- Controls speed of the game
            END IF;
         END LOOP;
         IF NOT Goodhit THEN
            Left := Left + 1;    -- right hand side missed, left wins point
            Get_Immediate(C, Avail);  -- toss any remaining character
            EXIT;
         END IF;
      -- right hand side made a good return, switch to right to left
         Speed := Duration(Float(Speed) * Speedup);  -- speed up the game
         Output := BlankOutput;

         FOR I IN REVERSE 1..Len-1 LOOP  -- send the ball back
            Output(I) := 'O';            -- moves right to left
            Output(I+1) := ' ';
            Put_Line(Output);
            IF I /= 1 THEN    -- last delay handled in GoodHit
               DELAY Speed;
            END IF;
         END LOOP;
         IF NOT Goodhit THEN     -- if left side missed
            Right := Right + 1;  -- right gets the point
            Get_Immediate(C, Avail);
            EXIT;
         END IF;
      -- Good hit, so return the ball, left to right
         Speed := Duration(Float(Speed) * Speedup); -- and speed up
         Output := BlankOutput;

      END LOOP;

      Put("Play again (y or n): ");    -- we get here if a shot was missed
      Get(C);
      EXIT WHEN C /= 'y' AND C /= 'Y';

   END LOOP;

   New_line;
   Put_Line("Thanks for playing Ping Pong");

END Pingpong;
```

Exercise: extend the game by using two specific keys (say A and L) for two players and alternating service from left and right every 5 games. Add scoring. Control the speed of return shots by slowing down for an early return (say in the first third of the period the ball is at the end of its travel) and speeding it up for a delayed return (say in the last third of the hit period).

16.3 Using the type Time from Calendar

We very often in writing games want to keep track of the time of day, or the time an event occurred, and the calendar package offers the types clock and time for this purpose. Let's return to the CritterCaretaker game and use time to set the critter's hunger and boredom levels. I will make the increase in hunger and boredom levels time dependent. To achieve this, I will add a third data member to the Critter class, the time the Critter was last played with, fed or listened to. Then, when the Caretaker interacts with the Critter, in the member function PassTime, I will calculate the amount of time that passed since the last interaction, update it, and increase the Critter's hunger and boredom levels by greater amounts the longer the Critter was left uncared for. Once the Critter's hunger level reaches 30, the Critter will die of starvation. PassTime will signal this by raising an exception HasDied. The main program will handle the exception by displaying a message and shutting down, otherwise this is the only change to it from the CritterCaretaker of Ch 10. The support package defining the critter class and its associated member data and subprograms also has minimal changes. The class has an additional data member, LastTime, and the member procedure PassTime now uses the elapsed time since the last player input to compute the hunger and boredom levels. Here is the code:

Main Program

```
WITH CritterTimerPak; USE CritterTimerPak;
WITH Text_IO; USE Text_Io;

PROCEDURE CritterTimer IS
   Crit : Critter;
   Choice : Character;
BEGIN
   Put_Line("Welcome to Critter Caretaker"); New_Line;
   Talk(Crit); -- Critter says Hi
   LOOP
      New_Line;
      Put_line("Type one character for what you would like to do with your
critter");
      Put( "E.at, L.isten, P.lay or Q.uit ? " );
      Get(Choice);
      CASE choice IS
        when 'E' | 'e' => Eat(Crit);
        when 'L' | 'l' => Talk(Crit);
        when 'P' | 'p' => Play(Crit);
        when 'Q' | 'q' => Exit;
        when others    => Put_line( "Invalid choice. Please try again with
E, L, P or Q." );
      END  case;
   END LOOP;

   Put("Bye-bye. Thanks for caring for me.");

EXCEPTION     -- only change to main – handle exception and exit
   WHEN HasDied =>   -- exception is defined in CritterTimerPak.ads
```

```
      Put_Line("Sorry, your Critter has died of starvation.");

END CritterTimer;
```

Package Spec

```
WITH Calendar; USE Calendar;

PACKAGE CrittertimerPak IS  -- package definition -  defines Critter type
   HasDied : exception;     -- define the exception to signal Critter died
   Slower : constant integer := 3;  -- bigger number slows Critter changes

   TYPE Critter IS PRIVATE;

   PROCEDURE Eat (Crit : IN OUT Critter; Food : Integer := 4);-- member
   PROCEDURE Talk (Crit : IN OUT Critter);    -- subprogram definitions
   PROCEDURE Play (Crit : IN OUT Critter; Fun : Integer := 4);

PRIVATE
   TYPE Critter IS
      RECORD
         Hunger  : Integer := 0;    -- data members
         Boredom : Integer := 0;
         lastTime : Time := clock;  -- new member: time of last visit
      END RECORD;
   FUNCTION GetMood(Crit : Critter) return Integer;
   PROCEDURE PassTime(Crit : IN OUT Critter; SomeTime : Integer := 1);

END CrittertimerPak;
```

Package Body

```
WITH Text_Io; USE  Text_IO;
PACKAGE BODY CritterTimerPak IS  --  Critter implementation

   FUNCTION GetMood(Crit : Critter) Return Integer IS
   BEGIN
      RETURN Crit.Hunger + Crit.Boredom;
   END;

-- Main change to the code is here, calc of increase in hunger and boredom
   PROCEDURE PassTime(Crit : in out Critter; SomeTime : Integer := 1) IS
      HaveWaited : CONSTANT Duration := Clock - Crit.LastTime; -- seconds
      Mult : integer := integer(HaveWaited)/Slower + 1;
   BEGIN
      Crit.Hunger := Crit.Hunger + mult*SomeTime;
      Crit.Boredom := Crit.Boredom + mult*SomeTime;
      IF Crit.Hunger > 30 THEN    -- if the Critter has not been fed for
         RAISE HasDied;           -- too long, signal it has died
      end if;
   END;

   PROCEDURE Talk(Crit : in out critter) IS
   Mood : integer;
   BEGIN
      PassTime(Crit);
      Mood := GetMood(Crit);
      Put("I'm a Critter and I feel ");
      CASE Mood IS
         WHEN 0..5 => Put_Line("Happy");
         WHEN 6..10 => Put_Line("Okay");
         WHEN 11..15 => Put_Line("Frustrated");
         when others => Put_line("Angry");
      END CASE;
-- Add a warning if the critter is starving
      If Crit.Hunger > 25 then put_line("and I am starving"); end if;
      Crit.LastTime := Clock;  -- update LastTime of interaction
   END;

   PROCEDURE Eat(Crit : in out Critter; Food : Integer := 4) IS
   BEGIN
      PassTime(Crit);
      Put_Line("Burrrp");
      Crit.Hunger := Crit.Hunger - Food;
      IF Crit.Hunger < 0 THEN
         Crit.Hunger := 0;
      END IF;
      Crit.LastTime := Clock;
   END;

   PROCEDURE Play(Crit : in out Critter; Fun : Integer := 4) IS
   BEGIN
      PassTime(Crit);
      Put_Line("Wheee");
      Crit.Boredom := Crit.Boredom - Fun;
      IF Crit.Boredom < 0 THEN
         Crit.Boredom := 0;
      END IF;
      Crit.LastTime := Clock;
   END;

END CritterTimerPak;
```

Now we have seen how time and duration work, let’s take a look at concurrency.

17 Concurrency

Since the very early days of computing, concurrency has played an important role in programming and much time and effort has been devoted to developing efficient ways of dealing with it. In games programming, there are often many things going on simultaneously and independently. The word for this is concurrency. In programming more advanced Games, you will inevitably be faced with programming concurrent activities. In simple games, like those we have looked at up till now, you do not need concurrency, but in many games it is convenient or even necessary to write the program as several parallel activities which cooperate when required. This is very much the case with multiple characters in game, games with multiple players and with multiple constraints, challenges, and interactions among characters. Concurrent programs can also exploit multiprocessor and multicore architectures directly.

In Ada, parallel activities are called tasks. In other languages the may be called processes, threads, jobs or activities. Each task is a sequential program and a concurrent system is composed of a number of tasks. The tasks appear to be executing in parallel, even on a single processor machine. Tasks really can execute in parallel on multiprocessor or multi-core machines.

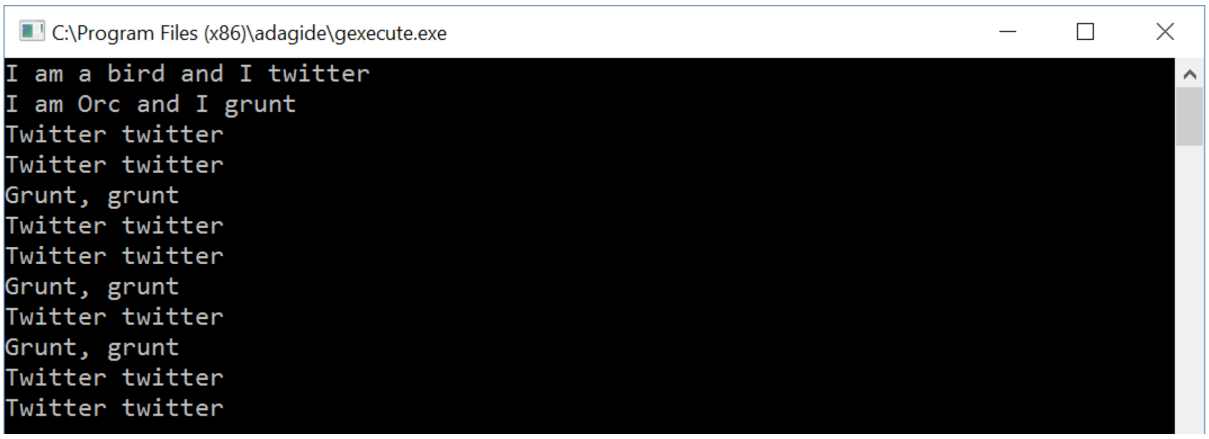
The tasks run independently and so are active, unlike subprograms which are passive and only run when called. Tasks can interact using synchronization with each other and they can communicate with each other where desired. Since they run independently, no assumptions can be made about the order in which they run and interact. You should think of each task as running on its own processor. They are activated automatically and so start running as soon as they are declared.

To illustrate, here are two tasks, one representing an Orc and the other Bird.

```
WITH Text_Io; USE Text_Io;

PROCEDURE Creatures IS

   TASK Bird;  -- one task, called bird

   TASK BODY Bird IS
      speed : duration := 1.0;
   BEGIN
      put_line("I am a bird and I twitter");
      LOOP
            DELAY speed;   -- slow it down so you can watch the bird move
            Put_line("Twitter twitter");  -- blank the previous position
      END LOOP;
   END Bird;

   TASK Orc;  -- one task, called bird

   TASK BODY Orc IS
      speed : duration := 2.0;
   BEGIN
      put_line("I am Orc and I grunt");
      LOOP
            DELAY speed;   -- slow it down so you can watch the bird move
            Put_line("Grunt, grunt");  -- blank the previous position
      END LOOP;
   END Orc;
```

```
BEGIN  -- The task is activated in parallel with the main prog.
   NULL;  -- The main program has nothing to do - it just waits
END Creatures;  -- for the tasks to finish.
```

As soon as you hit the run button, the two tasks both start running immediately and produce the following output. There is nothing to stop them, so they will keep running forever, or until the user stops them by closing the output window.


C:\Program Files (x86)\adagide\gexecute.exe

```
I am a bird and I twitter
I am Orc and I grunt
Twitter twitter
Twitter twitter
Grunt, grunt
Twitter twitter
Twitter twitter
Grunt, grunt
Twitter twitter
Grunt, grunt
Twitter twitter
Twitter twitter
```

17.1 Task Types

Let's now take a look at how a task can be treated as a type, i.e. a sort of class, and make multiple instances of the task. I will also set it up so some parameters can be passed to the task when an instance is created. These are called Discriminants and are limited to discrete and access types. I want to create a task type Creature, then create several creatures, an Orc, a Bird and a Dragon. I will pass to each instance of the task the name of the Creature, its greeting (both will be strings), the rate the greeting is repeated and how many times the task repeats before it terminates. Note that discriminants are not arguments as in subprogram calls, nor is there a return value as tasks do not return to the parent program. They continue to run by themselves, on their own merry way, until they terminate.

Here is the specification for the task type, which I will call Creature:

```
   TASK TYPE Creature(Name : NOT NULL ACCESS String;  -- Task Specification
      Greet : NOT NULL ACCESS String; -- NOT NULL --> can't be NULL
      Rate : Integer;
      Repeat : integer);  -- one task, called creature with 4 discriminants
```

When I make an instance of Creature, I will give it its name, greeting, rate and the number of times it runs. The name and greeting are strings, so are not discrete types, so I must use an access type and write:

```
       Troll_Name : aliased String := "Troll";
       TYPE A_Troll_Name IS ACCESS All String;
       This_Troll : A_Troll_Name := Troll_Name'Access;
```

Then I would use the access variable This_Troll when I activate the task by making an instance of it:

```
      Run_Orc    : Creature (This_Troll,  etc
```

I can, however, directly write more simply:

```
      Run_Orc   : Creature (new String'("Troll"),  etc
```

This implicitly defines the access type I need. The keyword **new** builds a new String "Troll" in the storage pool and returns a reference to it, which I can then pass to the task instance as a discriminant.

I also want to ensure that, when the task is activated, a valid string is passed to it. I can do this in the task specification by saying NOT NULL, as follows:

```
   TASK TYPE Creature(Name : NOT NULL ACCESS String; etc.
```

I want to tell the task how fast to repeat, but I can't use a duration as a discriminant, so I will use an integer for the rate and convert it to a duration when the task is activated.

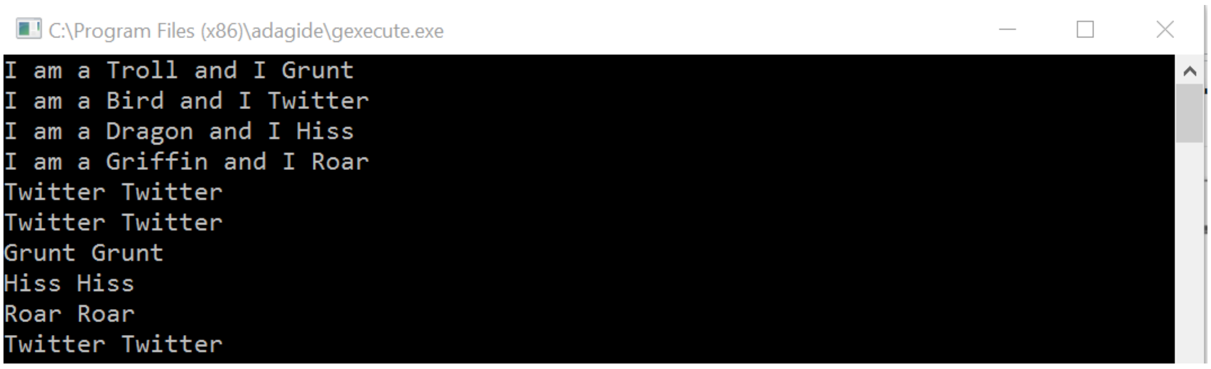
A point to note about Ada tasks is that they do not automatically report exceptions, so a task can have a silent death if a task raises an exception which is not handled by the task itself. So tasks should always be provided with an exception handler, otherwise, if something erroneous happens to crash the task, you will not get any report of what has gone wrong.

Finally, tasks, like all other objects in Ada, must have specifications and bodies:

Here is the code

```
WITH Text_Io; USE Text_Io;
WITH Ada.Exceptions; USE Ada.Exceptions;

PROCEDURE FourCreatures IS

   TASK TYPE Creature(Name : NOT NULL ACCESS String;  -- Task Specification
      Greet : NOT NULL ACCESS String; -- NOT NULL --> can't be NULL
      Rate : Integer;
      Repeat : integer);  -- one task, called creature with 4 discriminants

   type A_Creature is access Creature;  -- first declare access type
   Run_Griffin : A_Creature;  -- then a task access variable.

   task body Creature is
   speed : constant Duration := duration(Rate);
   begin
      Put_Line ("I am a " & Name.All & " and I " & Greet.All);
      for I in 1 .. Repeat LOOP
         DELAY Speed;
         Put_Line(Greet.All & ' ' & Greet.All);
      END LOOP;
   EXCEPTION   -- always provide tasks with an exception handler
      WHEN Error: OTHERS =>
         Put ("Unexpected exception: ");
         Put_Line (Exception_Information(Error));
   end Creature;

   Run_Orc    : Creature (new String'("Troll"),
                  new String'("Grunt"), 2, 5);   -- these 3 declarations
   Run_Bird   : Creature (NEW String'("Bird"),
                  NEW String'("Twitter"), 1, 8); -- make 3 instances of
   Run_Dragon : Creature (new String'("Dragon"),
                     new String'("Hiss"), 3, 4);    -- the task Creature

BEGIN  -- The 3 tasks above are activated in parallel with the main program
       -- Now we will make another one, dynamically, using the access type
       -- just to show how it is done
   Run_Griffin := NEW Creature (NEW String'("Griffin"),
                     NEW String'("Roar"), 3, 5);
END FourCreatures;  -- waits for the tasks to finish.
```

Here is a sample of the output:



```
I am a Troll and I Grunt
I am a Bird and I Twitter
I am a Dragon and I Hiss
I am a Griffin and I Roar
Twitter Twitter
Twitter Twitter
Grunt Grunt
Hiss Hiss
Roar Roar
Twitter Twitter
```

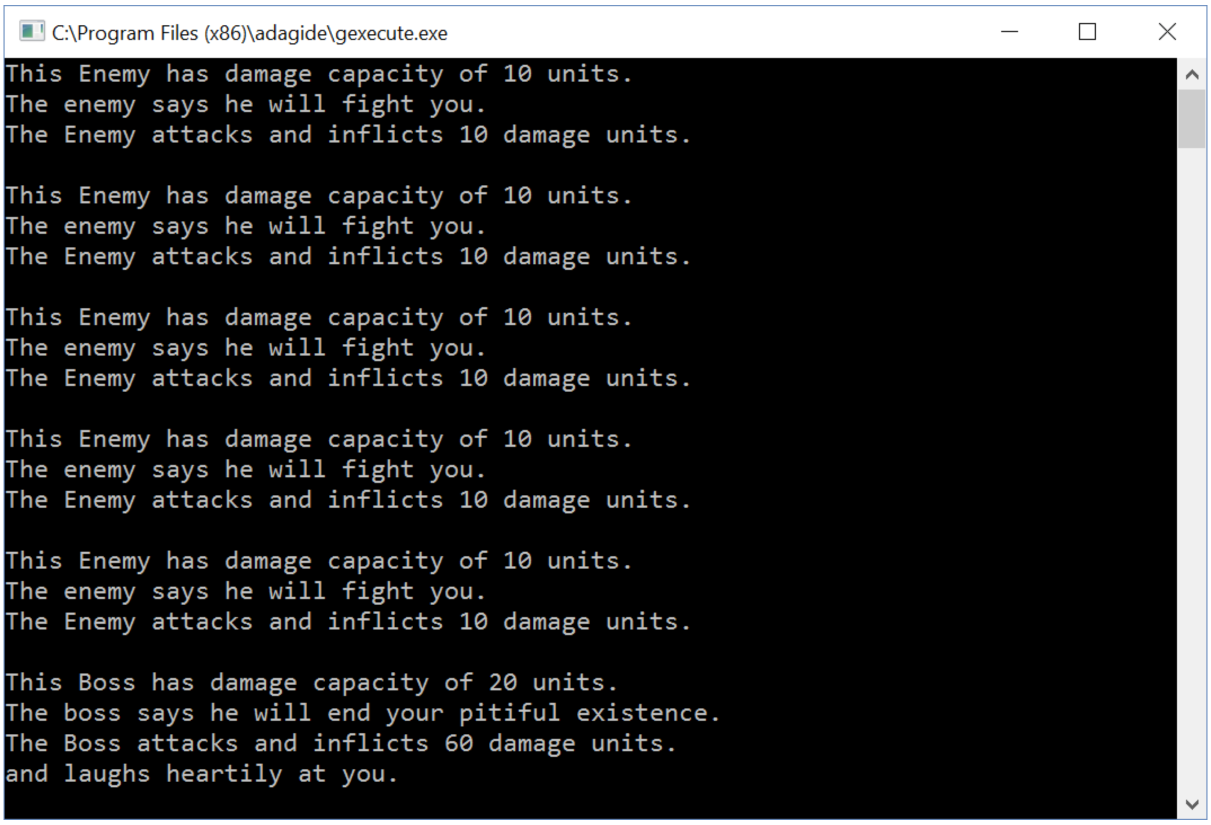
So we see that we can make task types and create multiple instances of them. Tasks can be made components of arrays or records, or parts of other objects. They can be used wherever a valid Ada type can be used, both built in or user-defined types. Some people maintain that tasks, being active, are the only true objects, but this is a discussion for another time.

17.2 Combining tasks with other objects

Now let’s see how we combine tasks with other objects. For this example I will return to the OvrBoss example, which uses the OvrBoss package to show attacks by enemies and by the Boss, who inflicts more damage. I can simply use the OvrBoss package and make an array of enemy tasks with it. Each one will then run independently and inflict damage. After a short delay to allow all the enemies to complete their attacks, the boss attacks to finish the job. Here is the code with 5 enemies:

```
with OvrBossPak; use OvrBossPak;
WITH Text_IO; USE Text_IO;
with ada.Exceptions; use ada.Exceptions;

procedure OvrBossTasks is

   TASK TYPE Enemies;

   TASK BODY Enemies IS
      ThisEnemy : Enemy := Make(10);
   BEGIN
      DisplayDamage(ThisEnemy);
      Taunt(ThisEnemy);
      Attack(ThisEnemy);
   EXCEPTION   -- always provide tasks with an exception handler
      WHEN Error: OTHERS =>
         Put ("Unexpected exception: ");
         Put_Line (Exception_Information(Error));
   end;

   type Enemy_Array IS ARRAY(1..5) OF Enemies;   -- make 5 enemies
   This_Enemy_Array : Enemy_Array;   -- as an array of 5 tasks

   AnEnemy : Enemy := make(10);      -- make one enemy, used by Boss
   ThisBoss : Boss := make(20, 3);   -- and one boss

BEGIN           -- all 5 enemies start executing automatically
   delay 0.2;  -- wait for all enemies to finish their attacks
   DisplayDamage(ThisBoss);   -- before the boss attacks
   Taunt(ThisBoss);
   Attack(ThisBoss, AnEnemy);

end OvrBossTasks;
```

And here is the output:



```
This Enemy has damage capacity of 10 units.
The enemy says he will fight you.
The Enemy attacks and inflicts 10 damage units.

This Enemy has damage capacity of 10 units.
The enemy says he will fight you.
The Enemy attacks and inflicts 10 damage units.

This Enemy has damage capacity of 10 units.
The enemy says he will fight you.
The Enemy attacks and inflicts 10 damage units.

This Enemy has damage capacity of 10 units.
The enemy says he will fight you.
The Enemy attacks and inflicts 10 damage units.

This Enemy has damage capacity of 10 units.
The enemy says he will fight you.
The Enemy attacks and inflicts 10 damage units.

This Boss has damage capacity of 20 units.
The boss says he will end your pitiful existence.
The Boss attacks and inflicts 60 damage units.
and laughs heartily at you.
```

Note that I have used the OvrBoss package as is with no modifications of any sort. The key to effective software design and reuse is to build packages that can be reused as is. Unfortunately, when we run this program the output sometime appears jumbled together. This is because the five enemy tasks have not been told to wait for each other when they output to the display. Since the display is a shared resource, we have to control the sharing.

17.3 Shared resources

Now let's take a look at another key issue in concurrent programming, that of using shared resources. This is often discussed in the context of shared data, where concurrent tasks read and write to common data and interfere with each other. There is much research in the area, especially in the field of databases. Fortunately, modern systems provide the means to handle shared resources properly, and so does Ada.

To illustrate the problem, suppose two tasks were both trying to print on the same printer, and their access to the printer was not managed properly, so the tasks both sent output to the printer at the same time. The output from the two tasks would be mixed up together, resulting in an unreadable mess. This is prevented on modern systems by only allowing one task to print at a time.

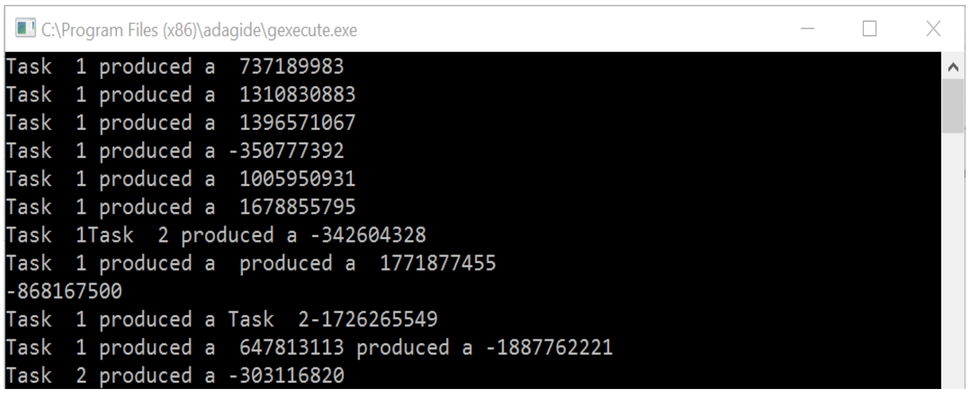
Now let’s look at an example of a task that writes random numbers to the display. I will make two instances of the task and when I run the tasks, let’s see what we get.

Here is the code:

```
WITH Text_Io; USE Text_Io;
WITH Ada.Exceptions; USE Ada.Exceptions;
WITH Ada.Numerics.Discrete_Random;

PROCEDURE SharedDisplay IS

   Repeat : Integer := 10;  -- the number of times the task runs
   PACKAGE Random_Int IS NEW Ada.Numerics.Discrete_Random (Integer);
   USE Random_Int; -- as in previous examples
   G : Generator;

   TASK TYPE DisplayNums(MyTaskNum : Integer);  -- one task
   TYPE DispNumsAcc IS ACCESS DisplayNums;
   D1, D2 : DispNumsAcc;

   TASK BODY DisplayNums IS
      Local : Integer;
   BEGIN
      FOR I IN 1 .. Repeat LOOP
         Local := Random(G); -- make and display a random number
         Put("Task "); put(integer'image(MyTaskNum)); put(" produced a ");
         Put(integer'image(Local));
         New_Line;
      END LOOP;
   EXCEPTION   -- always provide tasks with an exception handler
      WHEN Error: OTHERS =>
         Put ("Unexpected exception: ");
         Put_Line (Exception_Information(Error));
   END DisplayNums;

BEGIN
   Reset(G);  -- initialise the random number generator
   D1 := NEW DisplayNums(1); -- and start two tasks
   D2 := NEW DisplayNums(2);
END SharedDisplay;  -- waits for the tasks to finish.
```

And when the program is run, we might the output from the two tasks interfering with each other. As we expect, task D1 starts first and generates some clean output, but then D2 gets going and the output from the two tasks get mixed up together.

This is obviously not a desirable effect. What is more, every time you run the program, you get different interference, and sometimes none at all! If this were shared data, like your bank balance, and it was liable to be corrupted by tasks interfering with each other, you would not be pleased.



```
Task  1 produced a  737189983
Task  1 produced a  1310830883
Task  1 produced a  1396571067
Task  1 produced a -350777392
Task  1 produced a  1005950931
Task  1 produced a  1678855795
Task  1Task  2 produced a -342604328
Task  1 produced a  produced a  1771877455
-868167500
Task  1 produced a Task  2-1726265549
Task  1 produced a  647813113 produced a -1887762221
Task  2 produced a -303116820
```

Ada provides the means to share resources in a controlled way. There are several ways to do this. The first and simplest is to define the shared resource as **protected.** A protected resource can only be accessed by one task at a time, so the tasks cannot interfere with each other, other than a short delay while the task that has the resource finishes using it. Here I will use a protected subprogram to provide coordinated access to the shared resource, through calls on its visible protected operations

Here, the display is shared, so I will make all calls to the display output part of a protected procedure. To do this, I move all the text_Io calls into the protected procedure, which is

```
WITH Text_Io; USE Text_Io;
WITH Ada.Exceptions; USE Ada.Exceptions;
WITH Ada.Numerics.Discrete_Random;

PROCEDURE ProtectedSharedDisplay IS

   Repeat : Integer := 10;  -- the number of times the task runs
   PACKAGE Random_Int IS NEW Ada.Numerics.Discrete_Random (Integer);
   USE Random_Int; -- as per previous examples
   G : Generator;

   protected Display is  -- specify a protected type containing a
      procedure writeMsg(t, v : integer);  -- protected procedure
   end Display;

   protected body Display is  -- and write the protected code
      procedure WriteMsg(t, v : integer) is
      begin
         Put("Task "); put(integer'image(t)); put(" produced a ");
         Put(integer'image(v));
         New_Line;
      END WriteMsg;
   END Display;


   TASK TYPE DisplayNums(MyTaskNum : Integer);  -- one task
   TYPE DispNumsAcc IS ACCESS DisplayNums;
   D1, D2 : DispNumsAcc;

   TASK BODY DisplayNums IS
      Local : Integer;
   BEGIN
      FOR I IN 1 .. Repeat LOOP
         Local := Random(G); -- make and display a random number
```

```
            Display.WriteMsg(MyTaskNum, Local); -- call protected procedure
      END LOOP;
   EXCEPTION   -- always provide tasks with an exception handler
      WHEN Error: OTHERS =>
         Put ("Unexpected exception: ");
         Put_Line (Exception_Information(Error));
   END DisplayNums;

BEGIN
   Reset(G);  -- initialise the random number generator
   D1 := NEW DisplayNums(1); -- and start two tasks
   D2 := NEW DisplayNums(2);
END protectedSharedDisplay;  -- waits for the tasks to finish.
```

Now the output appears as it should – try it.

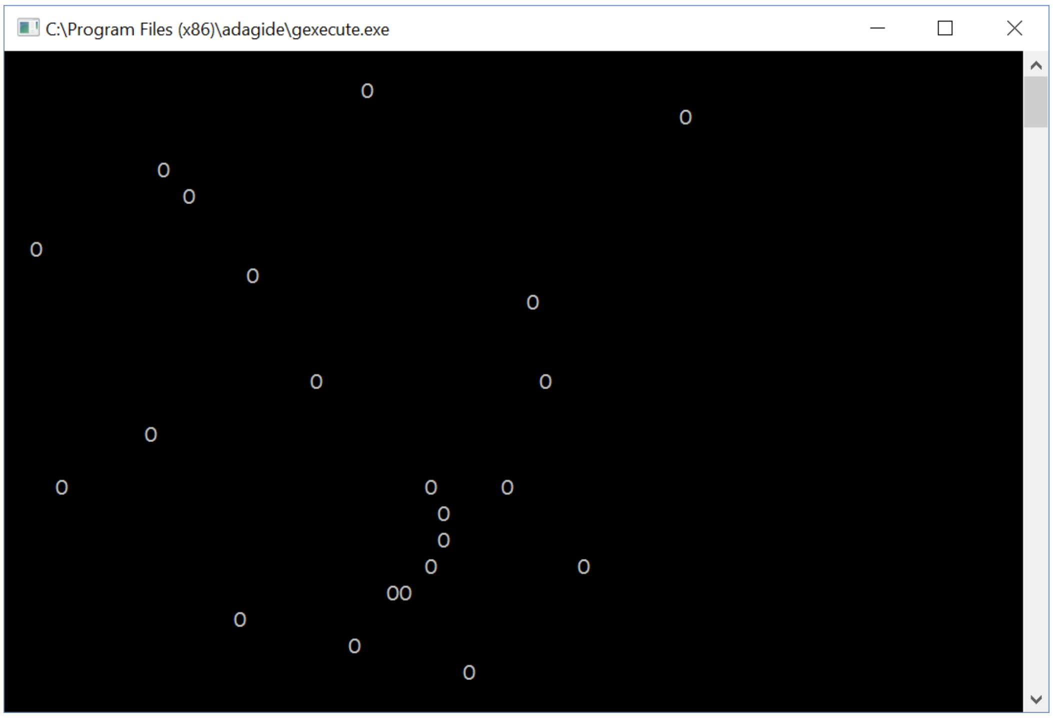
17.4 Bouncing balls

As a more interesting example, I will make a task that bounces a ball (the letter O) around the screen, starting in a random direction. Then I will make an array of five of the tasks so the 5 balls bounce around the screen together. The walls are just the edges of the screen and when the ball hits a vertical edge, it reverses its movement in the horizontal direction, and when it hits the top or bottom, it reverses its movement in the vertical direction.

Here is the code:

```
WITH Text_Io; USE Text_Io;
WITH Ada.Numerics.Float_Random; USE Ada.Numerics.Float_Random;
WITH Ada.Numerics; USE Ada.Numerics;
WITH Ada.Numerics.Elementary_Functions; USE
Ada.Numerics.Elementary_Functions;
WITH NT_Console; USE NT_Console;
WITH Ada.Exceptions; USE Ada.Exceptions;

PROCEDURE FiveBalls IS
   G : Generator;  -- random number generator in range 0.0..1.0
   TASK TYPE Ball;
   TASK BODY Ball IS
      PrevX, ThisX : Float := 40.0;  -- start in the centre of the screen
      PrevY, ThisY : Float := 12.0;  -- use floats for accurate positioning
      XPos, YPos  : Integer;   -- but the screen position must be integers
      Xvel, Yvel  : Float;  -- These hold the velocity of the motion
      Angle : Float;     -- and this is the starting angle
   BEGIN
      Reset(G);   -- start at a ramdom angle
      Angle := Random(G)*2.0*Pi;  -- radians (2*Pi = full circle)
      XVel := Sin(Angle);  -- convert to velocity in X, Y directions
      YVel := Cos(Angle);
      Goto_XY(40,12);      -- go to the starting position
      Put('O');
      LOOP
         DELAY 0.1;
         Goto_XY(Integer(PrevX), Integer(PrevY));
         Put(' ');    -- erase ball from previous position
         ThisX := PrevX + XVel;  -- move ball to new position
         XPos := Integer(ThisX); -- in X and Y directions
         ThisY := PrevY + YVel;
         YPos := Integer(ThisY);
         Goto_Xy(XPos, YPos);
         Put('O');               -- display ball in new position
         IF XPos = 1 OR XPos = 79 THEN  -- if hit a vertical wall
            XVel := -XVel;    -- reverse X velocity
         END IF;
         IF YPos = 1 OR YPos = 24 THEN  -- if hit a horizontal wall
            YVel := -YVel;    -- reverse Y velocity
         END IF;
         PrevX := ThisX;  -- save this position
         PrevY := ThisY;
      END LOOP;
   EXCEPTION   -- always provide tasks with an exception handler
      WHEN Error: OTHERS =>
         Put ("Unexpected exception: ");
         Put_Line (Exception_Information(Error));
   END Ball;

   TYPE BAT IS ARRAY (1 .. 5) OF Ball;  -- make an array
   BA : BAT;                          -- of 5 ball tasks

BEGIN  -- The task array is activated in parallel with the main prog.
   NULL;  -- The main program has nothing to do
END FiveBalls;
```

But when I run the program, I get a mess. Some runs are worse than others. Here is a bad one:



```
                            O
                                                     O

            O
              O

  O
                   O
                                         O


                        O                 O

           O

    O                            O     O
                                  O
                                  O
                                 O           O
                              OO
                  O
                           O
                                    O
```

Instead of 5 balls bouncing around cleanly, the tasks (as expected) interfere with each other when accessing the display, so the screen gets into a mess. This is another demonstration of the problem of concurrent access to a shared resource. In this case, the 5 tasks displaying the 5 balls share the display, and a task that goes to a position to erase a ball might be interrupted by a task writing a ball and erase the wrong position, or vice versa, write a ball in the wrong position

To see this problem in more detail, suppose one of the tasks, say task 3, has just called Goto_XY to move into position to erase a ball when another task, say task 1, also calls Goto_XY to move the screen position to display its ball. Then task 3 calls `Put(' ');` but this writes a blank character at the wrong screen position, so the actual ball that was meant to be erased is not and remains on the screen.

Managing access to shared resources in concurrent programming is a problem with a long history and here we see the problem clearly with the shared display. To protect shared resources, Ada currently favours the use of protected types. As the Ada Language Reference Manual (LRM), section 9.5, states: a protected object provides coordinated access to a shared resource, through calls on its visible protected operations, which can be protected subprograms or protected entries. A protected entry can have a guard, or entry barrier, to protect the resource when it is busy. In this example, I will add the protected resource example directly from the LRM, with no changes to the example code. This is shows a second method of protecting the shared resource.

```
WITH Text_Io; USE Text_Io;
WITH Ada.Numerics.Float_Random; USE Ada.Numerics.Float_Random;
WITH Ada.Numerics; USE Ada.Numerics;
WITH Ada.Numerics.Elementary_Functions;
USE Ada.Numerics.Elementary_Functions;
WITH NT_Console; USE NT_Console;
WITH Ada.Exceptions; USE Ada.Exceptions;

PROCEDURE FiveBalls1 IS

   G : Generator; -- random number generator in range 0.0..1.0

   PROTECTED TYPE Display IS  -- protect the shared display
      ENTRY Seize;       -- use entry call when a guard is used
      PROCEDURE Release; -- releases the display for other access
   PRIVATE
      Busy: Boolean := False; -- False when not busy
   END Display;

   PROTECTED BODY Display IS
      ENTRY Seize WHEN NOT Busy IS  -- guarded entry – open if True
      BEGIN                      -- executed when not busy is True
         Busy := True;          -- mark it as busy until released
      END Seize;

      PROCEDURE Release IS
      BEGIN
         Busy := False;   -- release the resource
      END Release;
   END Display;

   Screen : Display;

   TASK TYPE Ball;

   TASK BODY Ball IS
      PrevX, ThisX : Float   := 40.0; -- start in the centre of the screen
      PrevY, ThisY : Float   := 12.0;
      XPos, YPos   : Integer;
      Xvel, Yvel   : Float;
      Angle : Float;
   BEGIN
      Reset(G);
      Angle := Random(G)*2.0*Pi;  -- radians (2*Pi = full circle)
      XVel := Sin(Angle);  -- convert to velocity in X, Y directions
      YVel := Cos(Angle);
      Screen.Seize;       -- Seize the resource
      Goto_XY(40, 12);    -- display ball in initial position
      put('O');
      Screen.Release;     -- Then release it
      LOOP
         DELAY 0.1;
         ThisX := PrevX + XVel;  -- prepare to move ball to new position
         ThisY := PrevY + YVel;
         XPos := integer(PrevX);
         YPos := Integer(PrevY);
         Screen.Seize;
         Goto_XY(XPos, YPos);    -- erase previous position
         Put(' ');
         XPos := Integer(ThisX);
         YPos := Integer(ThisY);
         Goto_XY(XPos, YPos);    -- display ball in new position
         Put('O');
         Screen.release;
         IF XPos = 1 OR XPos = 79 THEN  -- if hit a vertical wall
            XVel := -XVel;    -- reverse X velocity
         END IF;
         IF YPos = 0 OR YPos = 24 THEN  -- if hit a horizontal wall
            YVel := -YVel;    -- reverse Y velocity
         END IF;
         PrevX := ThisX;  -- save this position
         PrevY := ThisY;
      END LOOP;
   EXCEPTION   -- always provide tasks with an exception handler
      WHEN Error: OTHERS =>
         Put ("Unexpected exception: ");
         Put_Line (Exception_Information(Error));
   END Ball;

   TYPE BAT IS ARRAY (1 .. 5) OF Ball; -- make an array
   BA : BAT; -- of 5 ball tasks

BEGIN  -- The task array is activated in parallel with the main prog.
   NULL;  -- The main program has nothing to do
END FiveBalls1;
```

Now the five balls bounce around the screen as they should – try it.

The key rule in concurrent programming is that statements that update a shared resource should not be interrupted: they should be indivisible. This applies to all shared resources, displays, printers, shared data, data files, etc. A sequence of statements that must appear to be executed indivisibly is called a critical section. The synchronisation required to protect a critical section is called mutual exclusion. The indivisible operation performed in the critical section is called an atomic operation. Critical sections should be designed to be as short as possible.

Ada's protected types provide efficient mechanisms for protecting shared data and shared resources. Protected types provide what computer scientists call a passive conditional mutual exclusion mechanism. A protected type can encapsulate shared data with all the operations to access that data. The operations have automatic mutual exclusion and guards (boolean conditions) can be placed on operations for condition synchronization. Tasks then access the shared data through the protected operations defined by the protected object. Protected functions provide read-only access to the protected data, while protected procedures provide read-write access to the protected data. Protected entries provide conditional read-write access to the protected data.

Only one protected read-write operation is permitted at a time, thus providing mutually exclusive access to the protected data, but there may be multiple read-only operations.

Protected function and protected procedure operations will run until they are completed so should be kept short.

17.5 Task interactions

The previous example used mutual exclusion to prevent the set of tasks from interfering with each other through an undesired interaction with the shared resource (the display). But what if we want the tasks to interact in a controlled way?

The next concept I will illustrate is that of task interaction by asking what happens if two balls collide. For now, I will extend the program to make a ball bounce back when it collides with another ball, so the tasks will interact by noting when two balls try to occupy the same position on the display. There are many ways to do this. I could use a vector or an array of occupied positions. I will choose to define an array corresponding to all the screen positions. Each array element will hold a Boolean that is true if the position is occupied and if a ball tries to move to that position, the move will be rejected and the ball will bounce back. I will do this by reversing the direction, calculating a new position and moving the ball away from the collision. Since this array will also be a shared resource, it must be protected, but this can be done with the same resource manager used for the display, since the array and the display will be updated together they can be in the same critical section. The array acts as a mediator between the tasks. Here is the code:

```
WITH Text_Io; USE Text_Io;
WITH Ada.Numerics.Float_Random; USE Ada.Numerics.Float_Random;
WITH Ada.Numerics; USE Ada.Numerics;
WITH Ada.Numerics.Elementary_Functions;
USE Ada.Numerics.Elementary_Functions;
WITH NT_Console; USE NT_Console;
WITH Ada.Exceptions; USE Ada.Exceptions;

PROCEDURE FiveBalls3 IS

   G : Generator; -- random number generator in range 0.0..1.0

   -- Make a boolean array for each position on the screen, plus
   -- a boundary around it for possible attempted moves off screen
   TYPE DisplayPosT IS ARRAY (0 .. 81, 0 .. 25) OF Boolean;
   DisplayPos : DisplayPosT := (OTHERS => (OTHERS => False));

   PROTECTED TYPE Display IS  -- protect the shared display and array
      ENTRY Seize;    -- use entry call when a guard is used
      PROCEDURE Release;-- releases the display for other access
   PRIVATE
      Busy: Boolean := False; -- False when not busy
   END Display;

   PROTECTED BODY Display IS
      ENTRY Seize WHEN NOT Busy IS-- guarded entry
      BEGIN               -- when it's not busy
         Busy := True;    -- mark it as busy until released
      END Seize;

      PROCEDURE Release IS
      BEGIN
         Busy := False;   -- release the resource
      END Release;

   END Display;

   Screen : Display;  -- make an instance of the Display type

   TASK TYPE Ball;

   TASK BODY Ball IS
      PrevX, ThisX, PrevY, ThisY           : Float;
      PrevXPos, PrevYPos, NewXPos, NewYPos : Integer;
      Xvel, Yvel, Angle                    : Float;
   BEGIN
      Reset(G);
      Angle := Random(G)*2.0*Pi;  -- radians (2*Pi = full circle)
      XVel := Sin(Angle);  -- convert to velocity in X, Y directions
      YVel := Cos(Angle);
      LOOP  -- find an unoccupied random position
         ThisX := Random(G) * 79.0 + 1.0;
         PrevX := ThisX;
         ThisY := Random(G) * 24.0 + 1.0;
         PrevY := ThisY;
         PrevXPos := Integer(ThisX);
         PrevYPos := Integer(ThisY);
         NewXpos := PrevXpos;
         NewYPos := PrevYPos;
         IF NOT DisplayPos(NewXPos, NewYPos) THEN
            Screen.Seize;  -- display ball in initial position
            Goto_XY(NewXPos, NewYPos);
            Put('O');
            DisplayPos(NewXpos, NewYPos) := True;
            Screen.Release;
            EXIT;  -- initial position found, so exit
         END IF;
      END LOOP;    -- otherwise try another position

      LOOP
         DELAY 0.2;
         ThisX := PrevX + XVel;  -- move to new position
         ThisY := PrevY + YVel;
         PrevXPos := Integer(PrevX);
         PrevYPos := Integer(PrevY);
         NewXPos := Integer(ThisX);
         NewYPos := Integer(ThisY);
         Screen.Seize;
         IF NewXPos /= PrevXPos OR NewYPos /= PrevYPos THEN -- test if
                                  --  actually will change position
            IF NOT DisplayPos(NewXPos, NewYPos) THEN  -- test if ok to move
               Goto_XY(PrevXPos, PrevYPos);    -- yes, move,
               Put(' ');      -- so erase previous position
               DisplayPos(PrevXpos, PrevYPos) := False; -- mark previous
                                           --  position as free
               Goto_XY(NewXPos, NewYPos);  -- display ball in new position
               Put('O');
               DisplayPos(NewXpos, NewYpos):= True;  -- mark new position
```

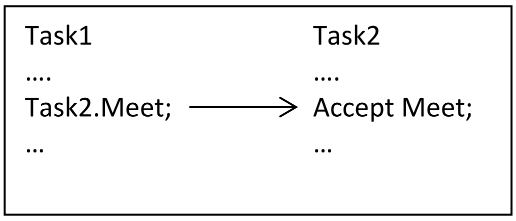
```
                                                            -- as occupied
            ELSE  -- reverse direction and move away
               XVel := -XVel;  -- reverse direction
               YVel := -YVel;
               ThisX := PrevX + XVel;  -- move ball to new position
               ThisY := PrevY + YVel;
               PrevXPos := Integer(PrevX);
               PrevYPos := Integer(PrevY);
               NewXPos := Integer(ThisX);
               NewYPos := Integer(ThisY);
               Goto_XY(PrevXPos, PrevYPos); -- erase previous position
               Put(' ');
               DisplayPos(PrevXpos, PrevYPos) := False;
               Goto_XY(NewXPos, NewYPos);   -- display ball in new position
               Put('O');
               DisplayPos(NewXpos, NewYpos):= True;
            END IF;
         END IF;
         Screen.Release;
         IF NewXPos = 1 OR NewXPos = 79 THEN  -- if hit a vertical wall
            XVel := -XVel;    -- reverse X velocity
         END IF;
         IF NewYPos = 0 OR NewYPos = 24 THEN  -- if hit a horizontal wall
            YVel := -YVel;    -- reverse Y velocity
         END IF;
         PrevX := ThisX;  -- save this position
         PrevY := ThisY;
      END LOOP;
   EXCEPTION   -- always provide tasks with an exception handler
      WHEN Error: OTHERS =>
         Put ("Unexpected exception: ");
         Put_Line (Exception_Information(Error));
   END Ball;

   TYPE BAT IS ARRAY (1 .. 5) OF Ball;-- make an array
   BA : BAT; -- of 5 ball tasks

BEGIN  -- The task array is activated in parallel with the main prog.
   NULL;  -- The main program has nothing to do
END FiveBalls3;
```

The difference in behaviour from the previous version is subtle, but with close observation you can see that when two balls collide, the second ball bounces back.

17.5 Intertask communications

In the above example, the 5 tasks bouncing balls communicate via a protected array, by writing to it and reading from it. This is a common approach and can also use containers. Sometimes, however, a game programmer will want two tasks to communicate directly, perhaps to synchronise, as when enemies synchronise their watches when preparing an attack, or to share critical information, like two spies exchanging secrets.

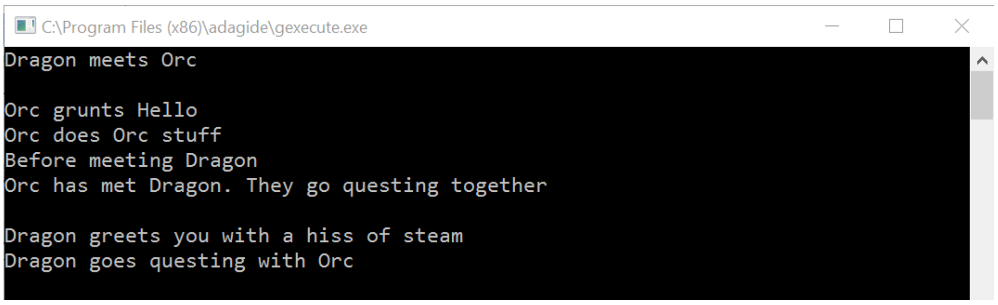
In Ada, tasks can communicate directly with each other via a Rendezvous. The Ada rendezvous synchronises the communicating tasks as well as (optionally) transferring data. Mailboxes, by comparison, are asynchronous. To do this, one task will make an Entry Call, using the same syntax as a procedure call with all the same parameter modes. The task being called must have an Accept statement, which includes built-in mutual exclusion to protect any data being shared.



An Entry call and corresponding Accept synchronises the two tasks. Whichever task reaches the rendezvous first, suspends and waits for the other task. When both are at the rendezvous, they can optionally exchange data, or just use the rendezvous to synchronise their activities.

As an example, suppose an Orc and a Dragon are represented by two tasks. Both do their own thing but then must meet so further action is coordinated. They could use a rendezvous for their meeting. To show this, I have written the following small example. Note that I use the existing package AbstractCreaturesPak I wrote back in Chapter 14 to define the Orc and the Dragon. Also, in this rendezvous, the Orc and Dragon tasks just synchronise, but do not exchange any information.

```
WITH Text_Io;
USE Text_Io;
WITH AbstractCreaturesPak;
USE AbstractCreaturesPak;

PROCEDURE OrcAndDragon IS

   AnOrc : Orc := Make ("Orc grunts Hello", 90);  -- from
AbstractCreaturesPak

   ADragon : Dragon := Make
      ("Dragon greets you with a hiss of steam", 100, TRUE);

   TASK Orc;

   TASK Dragon IS
      ENTRY Meet;        -- Entry point for Rendezvous
   END;

   TASK BODY Orc IS
   BEGIN
      Greet(AnOrc);      -- Library procedure from AbstractCreaturesPak
      Put_Line("Orc does Orc stuff");
      Put_Line("Before meeting Dragon");
      Dragon.Meet;  -- call rendezvous with Dragon
      Put_line("Orc has met Dragon. They go questing together");
   END Orc;

   TASK BODY Dragon IS
   BEGIN
      Put_Line("Dragon meets Orc");
      ACCEPT Meet;  -- accept rendezvous
      Greet(ADragon);
      Put_Line("Dragon goes questing with Orc");
   END Dragon;

BEGIN
   NULL;
END OrcAndDragon;
```



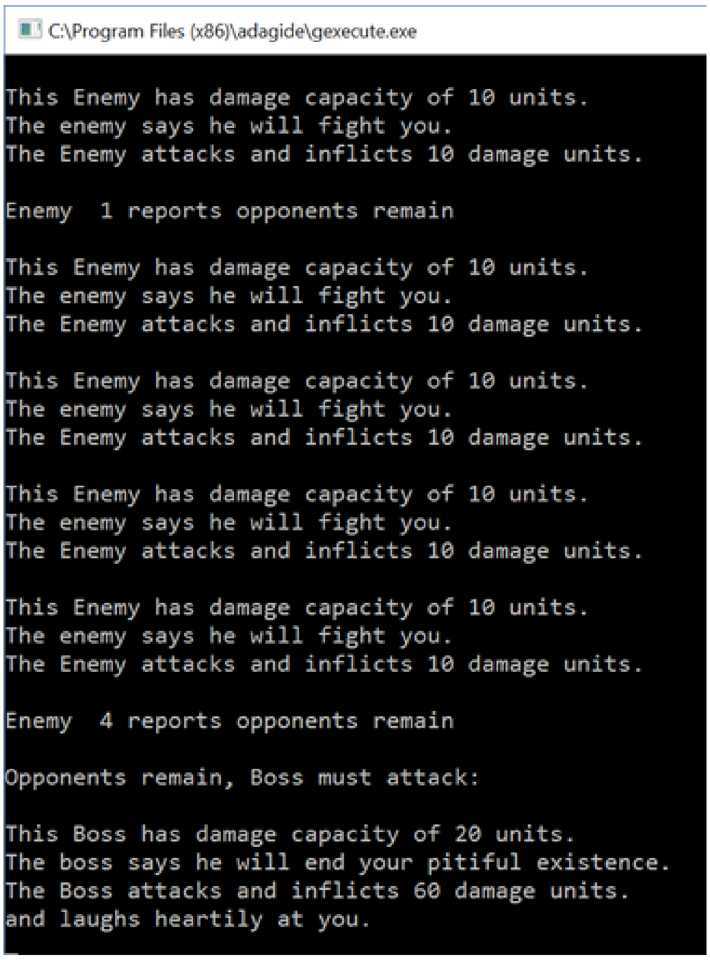
```
Dragon meets Orc

Orc grunts Hello
Orc does Orc stuff
Before meeting Dragon
Orc has met Dragon. They go questing together

Dragon greets you with a hiss of steam
Dragon goes questing with Orc
```

Let's suppose we want Orc to pass something to Dragon at their rendezvous, perhaps a magic Crystal, or one of his weapons. You can represent this as an item in an enumerated type listing the items encountered on the adventure. For now, for this simple example, I will just use a Boolean that, when true, represents the item Orc gives to Dragon.

```
WITH Text_Io; USE Text_Io;
WITH AbstractCreaturesPak; USE AbstractCreaturesPak;

PROCEDURE OrcAndDragon2 IS

   AnOrc : Orc := Make ("Orc grunts Hello", 90); -- in AbstractCreaturesPak

   ADragon : Dragon := Make
      ("Dragon greets you with a hiss of steam", 100, TRUE);

   TASK Orc;

   TASK Dragon IS
      ENTRY Meet(Gift : boolean);  -- Entry point for Rendezvous
   END;

   TASK BODY Orc IS
   BEGIN
      Greet(AnOrc);      -- Library procedure from AbstractCreaturesPak
      Put_Line("Orc does Orc stuff and finds a Magic Crystal");
      Put_Line("Before meeting Dragon");
      Dragon.Meet(True);  -- call rendezvous with Dragon
      Put_line("Orc has met Dragon. They go questing together");
   END Orc;

   TASK BODY Dragon IS
      HasCrystal : Boolean := false;
   BEGIN
      Put_Line("Dragon meets Orc");
      ACCEPT Meet(Gift : boolean) DO   -- accept rendezvous
         HasCrystal := Gift; -- and pass the data representing the gift
      END Meet;
      Greet(ADragon);
      IF HasCrystal THEN
         Put_Line("Dragon says thanks for the Magic Crystal");
      END IF;
      Put_Line("Dragon goes questing with Orc");
   END Dragon;

BEGIN
   NULL;
END OrcAndDragon2;
```

Orc gives Dragon the Magic Crystal by passing the parameter Gift during the Rendezvous. When information is exchanged at the rendezvous, the syntax is extended from simply Accept <entry>; (here Accept Meet;) to the form Accept <Entry(Parameters)> do <statements> End; Any valid name may be used for the Entry point and Ada types may be used for the parameters, including objects, containers, etc. Data may also be passed either way by setting the parameter mode to IN, OUT or IN OUT. The same rules apply as apply to subprogram parameters. Note, however, that the object being passed via the rendezvous is only visible during the actual rendezvous, from the Accept to the following End, so the object needs to be saved locally, as it will go out of scope once the rendezvous ends and no longer be accessible.

17.6 A bigger example of using rendezvous


C:\Program Files (x86)\adagide\gexecute.exe

```
This Enemy has damage capacity of 10 units.
The enemy says he will fight you.
The Enemy attacks and inflicts 10 damage units.

Enemy  1 reports opponents remain

This Enemy has damage capacity of 10 units.
The enemy says he will fight you.
The Enemy attacks and inflicts 10 damage units.

This Enemy has damage capacity of 10 units.
The enemy says he will fight you.
The Enemy attacks and inflicts 10 damage units.

This Enemy has damage capacity of 10 units.
The enemy says he will fight you.
The Enemy attacks and inflicts 10 damage units.

This Enemy has damage capacity of 10 units.
The enemy says he will fight you.
The Enemy attacks and inflicts 10 damage units.

Enemy  4 reports opponents remain

Opponents remain, Boss must attack:

This Boss has damage capacity of 20 units.
The boss says he will end your pitiful existence.
The Boss attacks and inflicts 60 damage units.
and laughs heartily at you.
```

In this example we will also use the OvrBoss package and will create a group of Enemy tasks, each of which will attack some of their opponents. Each Enemy will report to the Boss if they did not manage to kill all their opponents. When all the Enemies have reported, is any opponents remain, the Boss will attack to clean up the opponents that remain.

I will use a single Boolean variable, Neutralised, originally True, set to False if any opponents remain. A set of Enemies will be despatched, via an array of Enemies. Boss will then call the enemies one by one (Enemy one report, Enemy Two report, etc.) to ask for a report from each if any opponents remain. So each Enemy task will Accept a Report call and respond True if opponents remain, otherwise False.

Since all the Enemy tasks will also display their progress, the screen will again be treated as a shared resource so the Enemy messages will not interfere with each other.

Here is the code:

```
with OvrBossPak; use OvrBossPak;
WITH Ada.Numerics.Discrete_Random;
with text_io; use text_io;

PROCEDURE OvrBossReportTasks IS

   subtype prob is integer range 1..3;
   Package Achieved IS NEW Ada.Numerics.Discrete_Random(Prob);
   use Achieved;
   G : generator;

   Num_Enemies : constant integer := 5;

   PROTECTED TYPE Display IS  -- protect the shared display
      ENTRY Seize;     -- use entry call when a guard is used
      PROCEDURE Release;-- releases the display for other access
   PRIVATE
      Busy: Boolean := False; -- False when not busy
   END Display;

   PROTECTED BODY Display IS
      ENTRY Seize WHEN NOT Busy IS -- guarded entry
      BEGIN                  -- when it's not busy
         Busy := True;       -- mark it as busy until released
      END Seize;

      PROCEDURE Release IS
      BEGIN
         Busy := False;      -- release the resource
      END Release;
   END Display;

   Screen : Display;

   TASK TYPE Enemies IS
      ENTRY Report(Remain : out Boolean);
   END Enemies;

   TASK BODY Enemies IS
      ThisEnemy : Enemy := Make(10);
   BEGIN
      Reset(G);
      screen.seize;
      DisplayDamage(ThisEnemy);
      Taunt(ThisEnemy);
      Attack(ThisEnemy);
      screen.release;
      ACCEPT Report(Remain : OUT Boolean) DO
         Remain := (Random(G) = 1);
      END Report;
   end;

   type Enemy_Array IS ARRAY(1..Num_Enemies) OF Enemies;  -- make 5 enemies
   This_Enemy_Array : Enemy_Array;

   AnEnemy : Enemy := make(10);
   ThisBoss : Boss := make(20, 3);  -- and one boss
   Remain : Boolean;
   Neutralised : Boolean := True;

BEGIN
   FOR I IN 1..Num_Enemies LOOP   -- call for reports
      This_Enemy_Array(I).Report(Remain);
      IF Remain THEN              -- opponents remain
         new_line;
         Put_Line("Enemy " & Integer'Image(I)
                        & " reports opponents remain");
         Neutralised := False;    -- not completely neutralised
      END IF;
   END LOOP; -- wait for all enemies to report
   IF Not Neutralised THEN  -- not all neutralised
      new_line;
      put_line("Opponents remain, Boss must attack:");
      DisplayDamage(ThisBoss);  -- before the boss attacks
      Taunt(ThisBoss);
      Attack(ThisBoss, AnEnemy);
   ELSE
      Put_Line("No opponents remain, the Boss congratulates his troops");
   END IF;

END OvrBossReportTasks;
```

17.7 Dealing with alternative activities with the Select alternative.

Another key requirement in concurrent programming is the ability to coordinate several concurrent activities in an ordered fashion. There are several scenarios that can arise: time out a communication, communicate with multiple other tasks in any order, end a communication if the sender is not ready, terminate a communication or transfer control. Ada supports all these scenarios.

The format of the Select statement takes the form shown. The alternatives may be one or more Accept statements (for communications), a single delay statement (which must be the last in the list), an Else statement to terminate the select if no Accept statements can be triggered or a terminate to end the task if there is no active task that can communicate with our task. Whichever alternative is triggered first has its following code executed and then the select ends and the other alternatives are all abandoned.

```
SELECT
   Alternative 1
Or
   Alternative 2
Etc

End Select;
```

For example, a group of creatures might communicate with their carer by telling the carer if they are hungry, and then carer might then feed the creature. We can also add a delay as one of the alternatives, to do something else if no alternative is triggered. As an example, suppose we have a Dragon and a Pixie, which can ask to be fed, and if they don't ask for some period we want to check on them, we could then write code like that shown. Select structures like this are usually part of a loop, so the carer task is always ready to act.

```
Loop
   Select
      Accept PixieHungry;
      FeedPixie;
   Or
      Accept DragonHungry;
      FeedDragon;
   Or
      Delay SomeDelay;
      CheckCreatures;
   End Select;
 End Loop;
```

Here is a very simple example from the LRM of a select with a timeout to ensure appropriate action is taken in good time if required:

```
      select
         accept Driver_Awake_Signal;
      or
         delay 30.0*Seconds;
         Stop_The_Train;
      end select;
```

17.8 An example from StarTrek

In this example, the captain calls the chief engineer to report on the status of the warp drives. There are two tasks, **Monitor_Drive** and **Drive_Condition** that communicate with other, and which times out if there is no response from the drives. Usually there would be sensors for this, but here the drive sensor responses are simulated by user input. We would also expect the tasks to be doing other stuff as well to maintain the drive health, but here we focus only on the drive status.

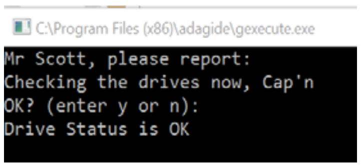
There are three points to note in the example. The first is the use of the abort statement to stop the Drive_Condition task. Abort allows one or more tasks to be shut down if required. The second point to note is that on non-real time platforms, text IO might not be interruptible, so a work around might be required. This is the case with windows 7, 8 and 10, so we make the input interruptible by using Get_Immediate. Finally, we use an entry point at the beginning of the Drive_Condition task to control precisely when it starts as it waits for a signal. This form of synchronisation is a basic use of the rendezvous, since you cannot depend for such synchronisation on assuming any particular order for which task runs first.

```
WITH Text_Io; USE Text_Io;

PROCEDURE Check_Warp_Drive IS

   Ch : Character;

   PROCEDURE Emergency_Shutdown IS
   BEGIN
      New_line(2);
      Put_Line("Drives fail to report.");
      Put_Line("Have to shut down, Cap'n.");
   END Emergency_Shutdown;

   TASK Monitor_Drive IS
      ENTRY Status ( OK : IN Boolean);
   END Monitor_Drive;

   TASK Drive_Condition IS
      ENTRY Start;
   END Drive_Condition;

   TASK BODY Drive_Condition IS
      Avail : Boolean;
   BEGIN
      Accept Start;
      Put_Line("Checking the drives now, Cap'n");
      Put("OK? (enter y or n): ");
      LOOP
         Get_Immediate(Ch, Avail); -- ensure this input
         EXIT WHEN Avail;          -- in interruptable
         DELAY 0.2;
      END LOOP;
      Monitor_Drive.Status(Ch = 'y');
   END Drive_Condition;


   TASK BODY Monitor_Drive IS
      Good : Boolean;
   BEGIN
```

```
            SELECT
               ACCEPT Status (OK : IN Boolean) DO
                  Good := OK;
               END Status;
               New_line;
               IF Good THEN
                  Put_Line("Drive Status is OK");
               ELSE
                  Put_Line("Drive status is not OK");
               END IF;
            OR
               DELAY 2.0;
               -- lost communication for more that 2 seconds
               ABORT Drive_Condition;
               Emergency_Shutdown;
            END SELECT;
         END Monitor_Drive;

      BEGIN
         Put_Line("Mr Scott, please report:");
         Drive_Condition.Start;
      END Check_Warp_Drive;
```

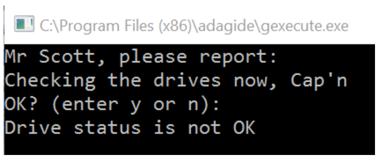
When the task is run, there are three possible outputs, for user inputs of y, n or nothing for 2 seconds.

Response: y



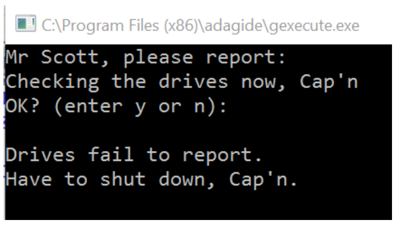
```
Mr Scott, please report:
Checking the drives now, Cap'n
OK? (enter y or n):
Drive Status is OK
```

Response: n



```
Mr Scott, please report:
Checking the drives now, Cap'n
OK? (enter y or n):
Drive status is not OK
```

No response



```
Mr Scott, please report:
Checking the drives now, Cap'n
OK? (enter y or n):

Drives fail to report.
Have to shut down, Cap'n.
```

17.9 More options for Select

The optional Else part in a select statement terminates the select if it is not selected immediately. For example, an adventurer needs a new weapon and if it is not immediately available the adventurer must retreat.

```
Select
   Accept SupplyWeapon;
   HaveWeapon := True;
Else
   Retreat;
End Select;
```

The Or Terminate option terminates the task if there is no other active task that can satisfy any of the Select entries.

```
Select
   Accept SupplyWeapon;
   HaveWeapon := True;
Or
   Terminate;
End Select;
```

The acceptance of any alternative can be made conditional by using a guard, which is Boolean precondition for acceptance. The accept is said to be open when the guard is true, as in this example:

```
select
   when not Filled =>
      accept Get(An_Item :
         in Item) do
         Data := An_Item;
      end Get;
      Filled := True;
 or
    when Filled =>
       accept Put(An_Item :
         out Item) do
         An_Item := Data;
       end Put;
       Filled := False;
 End select;
```

A program may also have a timed computation (different from a timed rendezvous). In this case the computation is aborted if it does not complete in the specified time. This called Asynchronous transfer of control.

```
Select
   WaitForSignal;
Then Abort
   HumongousComputation;
End Select;
```

The Signal to terminate can be a delay (ie a timeout) or an entry call (ie a rendezvous).

Here is an arithmetic drill game that uses this form of timing. The idea is for the computer to ask the user to enter the sum of two randomly chosen numbers and check if the answer is correct. This is the basis for many programs for practising arithmetic. In addition I will add a timeout, so if the answer is entered too slowly, the computer responds "Too slow".

A mixture of Ada and pseudocode for the core part of this game is

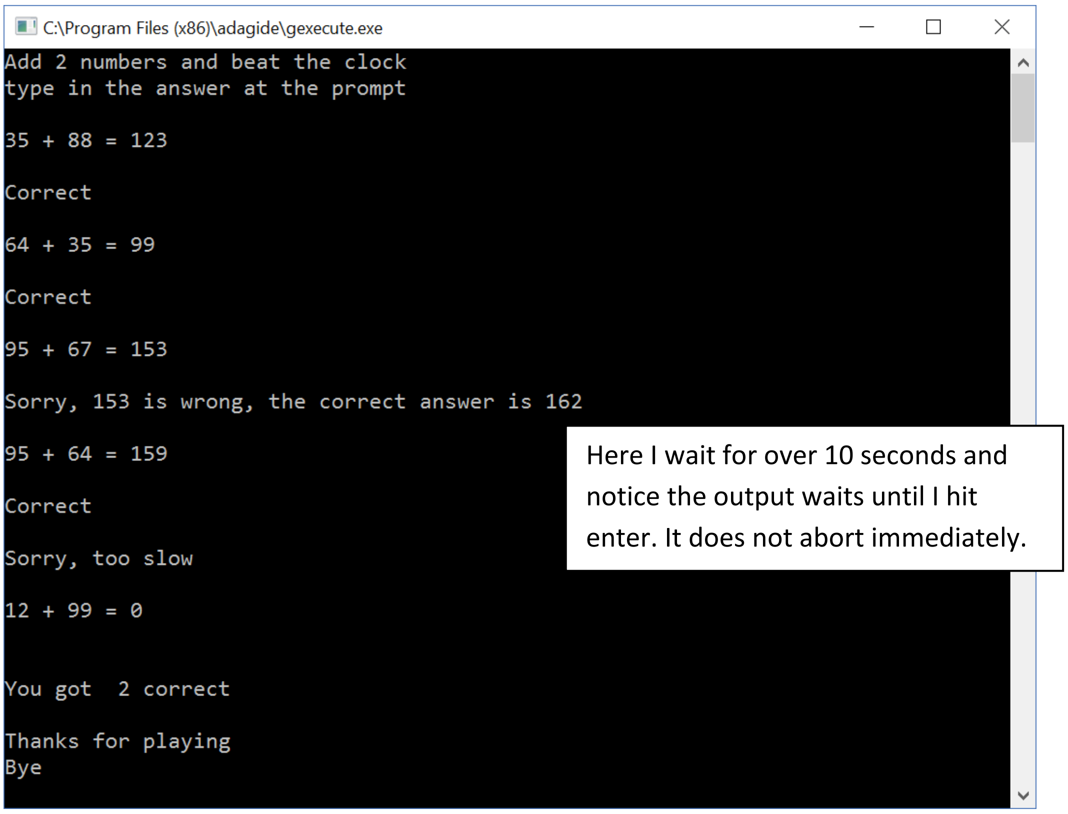
```
LOOP
   Generate two random numbers and display them
   SELECT
      DELAY 10.0;              -- give player 10 seconds to respond
      Put_Line("Sorry, too slow");
   THEN ABORT
      Get(Answer);             -- while waiting for the answer
      IF Answer is correct THEN
         Put_Line("Correct");
      ELSE
         Put_Line("Sorry, incorrect");
      END IF;
   END SELECT;
END LOOP;
```

I will expand the code by allowing a response of 0 to end the program, and the usual preambles, greetings and sign off at the end. I will also add code to keep track of the number of correct answers given. Here is the complete code.

```
WITH Text_Io; USE Text_Io;
WITH Ada.Integer_Text_Io; USE Ada.Integer_Text_Io;
WITH Ada.Numerics.Discrete_Random;

PROCEDURE AddQuick1 IS
```

```
   SUBTYPE Vals IS Integer RANGE 10..99;
   PACKAGE Random_Val IS NEW Ada.Numerics.Discrete_Random (Vals);
   USE Random_Val;  -- generates random integers in range 10..99
   G : Generator;

   First, Second, Ans, Numright : Integer;

BEGIN
   Reset(G);
   Numright := 0;
   Put_Line("Add 2 numbers and beat the clock.");
   Put_Line("Type in the answer at the prompt.");
   Put_Line("Enter zero to end.");
   New_Line;
   LOOP
      First := Random(G);  Second := Random(G);
      Put(First, Width => 2); Put(" + ");
      Put(Second, Width => 2); Put(" = " );
      SELECT
         DELAY 10.0;
         New_Line;
         Put_Line("Sorry, too slow");
      THEN
         ABORT
         Get(Ans); New_Line;
         IF Ans = First + Second THEN
            Put_Line("Correct");
            Numright := Numright + 1;
         ELSIF Ans /= 0 THEN
            Put("Sorry, "); Put(Ans,2);
            Put(" is wrong, the correct answer is ");
            Put(First+Second, Width => 2);
            New_Line;
         END IF;
      END SELECT;
      EXIT WHEN Ans = 0;
      New_Line;
   END LOOP;

   New_Line;
   Put("You got "); Put(Numright, Width => 2); Put_Line(" correct");
   New_Line; Put_Line("Thanks for playing"); Put_Line("Bye");
END AddQuick1;
```

Running this program under Microsoft Windows 7, 8 or 10 gives output like this:



```
Add 2 numbers and beat the clock
type in the answer at the prompt

35 + 88 = 123

Correct

64 + 35 = 99

Correct

95 + 67 = 153

Sorry, 153 is wrong, the correct answer is 162

95 + 64 = 159

Correct

Sorry, too slow

12 + 99 = 0


You got  2 correct

Thanks for playing
Bye
```

Here I wait for over 10 seconds and notice the output waits until I hit enter. It does not abort immediately.

The slow response is not correct, but the program does not abort the input, which it was expected to do. If you are running on Windows, then the real-time Annex D is not fully supported, since Windows does not provide a safe way to terminate a thread immediately without the thread's cooperation. These problems can sometimes be fixed in some cases by using Pragma Polling(ON); (for more information, see <http://docs.adacore.com/gnat_rm-docs/html/gnat_rm/gnat_rm/implementation_defined_pragmas.html#pragma-polling> but it does not help in this example.

A possible solution is to collect the input in a loop, one character at a time using Get_Immediate, saving the characters in a string with a short delay between the calls to get_immediate, then converting the collected string internally. To do this, replace the statement Get(Ans) in the above code with a function call Ans := GetAns; with the function code body as follows:

```
Function body GetAns is
   Inp  : String := "             ";
   EOL  : CONSTANT Character := Character'Val (13); -- end of line
   I    :          Integer   := 1;
   Ch   :          Character := ' ';
   More :          Boolean;
BEGIN
   LOOP
      Get_Immediate (Ch, More);
      EXIT WHEN Ch = EOL;
      IF More THEN
         Put (Ch);
         Inp(I) := Ch;
         I := I + 1;
      ELSE
         DELAY 0.1;
      END IF;
```

```
        END LOOP;
        Return Integer'Value(Inp(1..I));
    END GetAns;
```

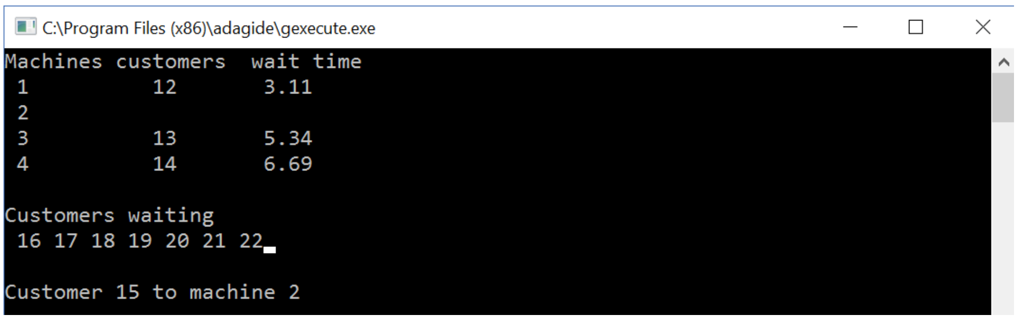
The loop with the delay 0.1 allows the input to be tested every 0.1 seconds and pre-empted so the Abort works as expected and the input is terminated as soon as the timeout expires. The complete code is provided in AddQuick2.adb.

This program can easily be expanded to allow exercises in other arithmetic like subtraction or multiplication, and to adjust the level of difficulty by using smaller or larger numbers. It would also be useful to add input error checking, i.e. that the input characters may only be digits. These are left to you as exercises.

17.10 A games room queue manager simulation using tasking

We now know enough about Ada tasking to write a more ambitious example. In this example, we will build a queue manager, like the game lobby, but with multiple services. Imagine a games arcade with multiple games machines and a queue of customers waiting to play. When a games machine is free, the customer at the head of the queue get to use the games machine. In this example, we have customers arrive at random intervals, and use the games machines for random times. The simulation displays the customer queue and games machine usage.

Here is a typical output after the simulation has run for a while:



```
Machines customers  wait time
 1          12       3.11
 2
 3          13       5.34
 4          14       6.69

Customers waiting
 16 17 18 19 20 21 22

Customer 15 to machine 2
```

As can be seen, machine 2 is vacant and customer 15 has been asked to go to machine 2, has left the queue and will get to machine 2 shortly.

This code is a useful framework for a more general game lobby or in fact any kind of customer queueing system. It can easily be extended to handle more machines, machines of different types or multiple queues.

Let’s think how we could design a simulation like this. After some thought, I decided the best approach was simply to follow what a customer would when entering the games arcade. So we have customers, who arrive at the games arcade, join the queue, wait for a vacant games machine, play for a while, then leave.

From this we can see that we have the following objects: customers, the games arcade that contains a number of games machines on which customers play. Perhaps we also need a machine manager, to send customers to free machines. We want to show all this on the display, so I will make a protected object, display, for this. So we have several objects: the arcade, machines, machine manager, customers and display. Perhaps we also need a queue, but for now let’s suppose each customer, on arrival at the arcade, gets issues a ticket. Then the ticket number can be used for the customer's position in the waiting queue, to call the customers to a machine when one is free and to free the machine and leave the arcade after playing.

Let's work through the code. We start with the usual preamble, "with" the packages we need, including the float random number generator, which returns numbers in the range 0.0 .. 1.0. We specify the maximum number of customers, the number of machines and the simulation time scale, so we can vary all of these easily. It is good practice to make numbers like these named constants and then use them throughput the program.

So here is the preamble:

```
with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
with Ada.Float_Text_Io; use Ada.Float_Text_Io;
with Ada.Numerics.Float_Random; use Ada.Numerics.Float_Random;
with Ada.Calendar; use Ada.Calendar;
with screen; use screen;
     -- DC Levy, May 2017: *** before compiling ***
     --   on Unix/Linux rename body ansiscreen.adb to screen.adb
     --   on Win2000/NT/XP/7/8/10, rename body ntscreen.adb to screen.adb
     --      and make sure you have the package NT_Console in the
     --      same directory.

procedure Queue_manager is

   G : Generator;
   Number_Of_Customers : constant := 20;  -- length of simulation
   Number_Of_Machines : constant := 4;
   Ts : constant := 1.0;  -- simulation time scale, increase to slow down
   Service_Time : constant := 12.0;         -- maximum value
   Interarrival_Time : constant := 2.0;     -- maximum value
```

Now let's look at the customers.

The customer instance, once launched, must:

Get a ticket, which will be the customer's position in the queue
Join the queue waiting for a machine, noting the time
Requests a machine. Waiting customers are automatically queued
When a machine is available, the customer at the front leaves the queue
The customer moves to the free machine
The customer plays for a while (in this simulation, this is represented just by a delay)
When done, the customer leave the machine
The customer exits the arcade

The above actions can all be set up as calls to the Arcade object, which I will consider next. For now, we specify a customer as:

```
   task type Customer is  -- simulates a customer
   end Customer;

   type A_Customer is access Customer;
```

Then later, in the Main program, we can create customers using:

```
   Next_Customer : A_Customer;    -- use this to create customers
```

Now lets look at the body of the Customer task:

```
   task body Customer is
      Machine_Id, My_Ticket: Positive;
      Enter_At : Time; Wait_For : Duration;
   begin
      Arcade.Enter(My_Ticket);  -- enter Arcade and get ticket
      Enter_At := Clock;
      Display.Add_Customer(My_Ticket);
      delay Ts*(0.5);          -- goes to queue and waits a bit
      Machine_Mgr.Req_Machine(Machine_Id);       --asks for a Machine
      Machine_Array(Machine_Id).Service_Call;  -- goes to Machine
      Wait_For := Clock - Enter_At;  -- time in queue
      Arcade.Leave_Queue;
      Display.Move_To_Machine(My_Ticket, Machine_Id, Wait_For);
      Machine_Array(Machine_Id).Service_Complete; -- waits till done
      Arcade.Leave_Arcade(Wait_For);    -- exit Arcade, show time in queue
      Display.Remove_Customer(Machine_Id);
   end Customer;
```

The next item is the Arcade. The Customer calls the Arcade to Enter and get a ticket and join the queue, Leave the Queue and Leave the Arcade. So we can specify the Arcade as a protected type, and keep track of the tickets, Number of Customers and the number in the Queue. We will also record the maximum number of customers ever in the Arcade and the longest time the Customers wait in the queue, for later reporting. The code is as follows:

```
   protected Arcade is  -- track no of customers and issue tickets
      procedure Enter(Ticket : out Positive);
      procedure Leave_Arcade(Wait : Duration);
      procedure Leave_Queue;
      function Num_In_Queue return Natural;
      function Max return Natural;
      FUNCTION Max_Wait RETURN Duration;
      entry IsEmpty;
   private
      Next_Ticket : Natural := 0;   -- ticket counter
      Numberofcustomers : Natural := 0;  -- number in the Arcade
      In_Queue : Natural := 0;
      Maxever : Natural := 0;  -- most ever in the Arcade
      Longest_Wait : Duration := 0.0;
   end Arcade;

   protected body Arcade is
```

Procedure Enter returns the next ticket number to the caller and keeps track of the data:

```
      procedure Enter(Ticket : out Positive) is begin
         Next_Ticket := Next_Ticket + 1;
         Numberofcustomers := Numberofcustomers + 1;
         In_Queue := In_Queue + 1;
         if Numberofcustomers > Maxever then
            Maxever := Numberofcustomers;
         end if;
         Ticket := Positive(Next_Ticket);
      end;
```

Leave_Arcade decrements the Number of Customers in the Arcade and udates Longest_Wait:

```
      procedure Leave_Arcade(Wait : Duration) is begin
         Numberofcustomers := Numberofcustomers - 1;
         if Wait > Longest_Wait then
            Longest_Wait := Wait;
         end if;
      end;
```

The rest of the procedures in Arcade are vey simple:

```
      procedure Leave_Queue is begin
         In_Queue := In_Queue - 1;
      end;

      function Num_In_Queue return Natural is begin
         return In_Queue;
      end;

      function Max return Natural is begin
         return Maxever;
      end;

      function Max_Wait return Duration is begin
         return Longest_Wait;
      END;

      ENTRY IsEmpty WHEN NumberOfCustomers = 0 IS begin
         NULL;         -- caller is suspended until the Guard is True
      end IsEmpty;

   end Arcade;
```

Now we can think about the machines. When a machine becomes free, the customer at the head of the queue goes to that machine. The machine manager will manage this and will be discussed next. The machine will keep its number, accept a customer's service call and a signal when the customer is finished and the service is complete. We will define a machine task and then make a group of machines as an array. Later we will activate each machine in the main program.

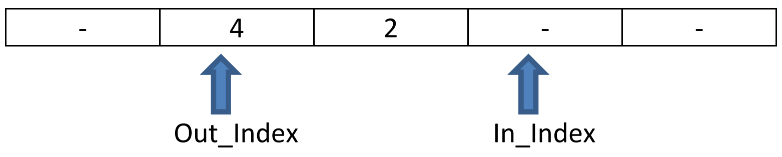
```
   type Machine_Pool_Type is array(1 .. Number_Of_Machines) of Positive;

   task type Machine is   -- Machine get allocated and free themselves
      entry Your_Num_Is(I : Positive);
      entry Service_Call;
      entry Service_Complete;
   end Machine;

  task body Machine is   -- serves customers by keeping them waiting!!
      My_Num : Positive;
   begin
      accept Your_Num_Is(I : Positive) do
         My_Num := I;
      end Your_Num_Is;
      delay 0.5; -- give me time to take up my position
      loop
         select   -- customers can now call for the machine
         accept Service_Call;    -- accept a customer
         delay Duration(Ts*(Random(G) * Service_Time)); -- at machine
         accept Service_Complete;  -- customer is finished playing
         delay Duration(Ts*(Random(G) * 2.0));  -- do other stuff a bit
         Machine_Mgr.Free(My_Num);   -- signal the manager I am done
      or
         terminate;    -- when there are no more customers
      end select;
   end loop;
end Machine;

Machine_Array : array(1..Number_Of_Machines) of Machine;
```

The Machine Manager then accepts requests for machines, and signals from machines when their current customers finish playing and free the machines. The manager will keep a list of free machines, allocate machines on request and return machines to the list of free machines when a customer is done. This is a kind of list called a bounded buffer, or a circular queue,

17.10.1 The Bounded Buffer

The bounded buffer is a classic data structure that is used in many applications, usually for buffering data. Because it is so common, it is worth understanding in detail, which is why I have put this discussion in a separate section. Here I use the bounded buffer structure for the machine manager, showing how data structures can often be used for many applications outside of their traditional use. It is always useful to think outside of the box.

There are many example implementations that can be found on the web. My solution is based on the example in John Barnes's book on "Programming in Ada 2012", which implements a bounded buffer as a protected type. In our case, the buffer will be an array of integers, equal in size to the number of games machines in the arcade, initially loaded with the numbers 1, 2 .. maxMachines. The numbers are then allocated as customers request machines and are returned to the buffer when machines become free. Since the customers use the machines for random amounts of time, the machine numbers are returned in random order. Suppose we have 5 machines. Then at some point the buffer might contain this, indicating that machines 4 and 2 are free:

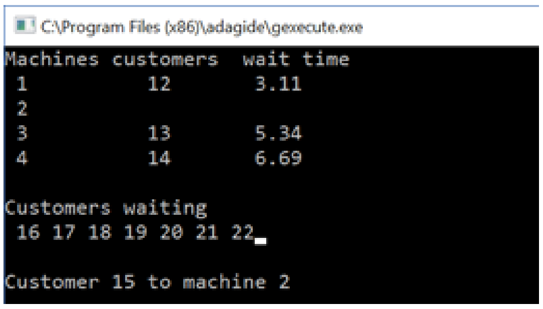


The objective is to allow items to be added to and removed from the buffer in a first-in–first-out manner in any order, but to prevent the buffer from being overfilled or under-emptied. This is done by the introduction of protected entries which always have barrier conditions. When a machine becomes free, its number returned to the buffer, into the cell pointed to by In_Index and In_Index is incremented so it points to the next empty cell and Count is incremented to keep track of the number of machines that are free. When one is requested its number is read from the buffer, from the cell pointed to by Out_Index, Out_Index is incremented and Count is decremented. When either In_Index or Out_Index reach the end of the buffer, they are wrapped around to point back to the first entry, using the Mod function. Count provides the barrier conditions – when Count = 0 the buffer is empty so no machine is free, and when Count = maximum number of machines, the buffer is full.

A request for a machine's number always gives the machine at the head of the buffer, and machines' numbers can be returned to the end of the buffer in any order. If the buffer is empty, it means all the machines are busy, a request from a customer for a machine will automatically be queued until one is freed and returned to the buffer. If it is full, all the machines are free. So in the main program we start by filling the buffer with the machine numbers 1 .. NumberOfMachines, after which requests can be sent for machines by the customers.

```
protected Machine_Mgr is                 -- Machine allocator
   entry Req_Machine (C : out Positive);  -- get a Machine
   entry Free (C : in Positive);    -- free a Machine
private
   Machine_Pool : Machine_Pool_Type; -- defines above.
   Count      : Natural := 0;
   In_Index   : Positive := 1;
   Out_Index  : Positive := 1;
end Machine_Mgr;

protected body Machine_Mgr is
   entry Free(C : in Positive)      -- free a Machine
         when Count < Machine_Pool'Length is
   begin
      Machine_Pool(In_Index) := C;
      In_Index := (In_Index mod Machine_Pool'Length) + 1;
      Count    := Count + 1;
   end Free;

   entry Req_Machine (C : out Positive)   -- get a Machine
         when Count > 0 is
   begin
      C := Machine_Pool(Out_Index);
      Out_Index := (Out_Index mod Machine_Pool'Length) + 1;
      Count     := Count - 1;
   end Req_Machine;
end Machine_Mgr;
```

Now we'll think about the customer queue. As mentioned above, when a customer arrives and requests a machine from the machine manager, if machines are available, the customer gets one, otherwise the machine manager queues the customer's request and this forms the customer queue, so no other code is needed to manage the queue (hooray!).



```
Machines customers  wait time
 1         12        3.11
 2
 3         13        5.34
 4         14        6.69

Customers waiting
 16 17 18 19 20 21 22

Customer 15 to machine 2
```

Let's now look at the display. Here it is again. We see that we will display a heading, and list the machines, then, as the program runs, we show the customers in the queue, the customer at each machine the time the customer waited in the queue. Customers get added to the back of the queue and moved to a machine from the front of the queue. We need routines for each of these actions, plus of course, since there are multiple tasks running and sharing the display, it must be a protected resource to avoid interference. We will specify it as a protected type:

```
   type Queue_Pos is (Back, Front);

   protected Display is
      procedure Add_Customer(Ticket : Positive);  -- to back
      procedure Move_To_Machine(Ticket : Positive; -- from front
         Machine_Id : Positive; In_Queue : Duration);
      PROCEDURE Remove_Customer(From_Machine : Positive);
      Procedure Write_Final_Msg;
   private
      P : Position; -- Defined in package Screen
   end Display;
```

The display operations do not affect the Customers, the Machines, or their data. They only present the information in a format that makes it easy to see what is happening. This requires that the data gets written to the correct positions on the screen, for which I again use the MoveCurson function in the Screen package, AnsiScreen in Unix/Linux/Solaris or Gerry van Dijk's NT_screen in Windows.

```
   protected body Display is

      procedure Write_Queue(Ticket : Positive; Where : Queue_Pos) is
         Waiting : Natural;
      begin
         P.Row := Number_Of_Machines + 4; P.Column := 1;  -- display queue
         Movecursor(P);
         Waiting := Arcade.Num_In_Queue;
         for I in 1 .. Waiting loop
            if Where = Back then Put(Ticket-Waiting+I, Width => 3);
            else Put(Ticket+I-1, Width => 3);
            end if;
         end loop;
         if Where = Front then
            Put("        "); -- delete possible trailing customers
         end if;
      end Write_Queue;

      procedure Add_Customer(Ticket : Positive) is  -- adds to queue
      begin
         Write_Queue(Ticket, Back);
      end Add_Customer;

      procedure Move_To_Machine(Ticket : Positive;
            Machine_Id : Positive; In_Queue : Duration) is
      begin
         P.Row := Number_Of_Machines + 6; P.Column := 1;
         Movecursor(P);
         put("Customer "); put(ticket, 1);
         put(" to machine "); put(Machine_Id, 1);
         delay 0.2;
         Write_Queue(Ticket+1, Front);  -- removes from front of queue
         P.Row := 1 + Machine_Id;    -- display customer at Machine
         P.Column := 10;
         Movecursor(P);
         Put(Ticket, Width => 5);
         Put(Float(In_Queue), Fore => 8, Aft => 2, Exp => 0);
      end Move_To_Machine;

      procedure Remove_Customer(From_Machine : Positive) is
      begin
         P.Row := 1 + From_Machine;  -- leaves Machine
         P.Column := 10;
         Movecursor(P);
         put("                "); -- erase entry against Machine
      end Remove_Customer;

      PROCEDURE Write_Final_Msg IS
      BEGIN
         P.Row := Number_Of_Machines + 9; P.Column := 1; Movecursor(P);
         Put_Line("Arcade is closed");
         Put("Max number of customers in Arcade was ");
         Put(Arcade.Max, Width => 3); New_Line;
         Put("Longest time in queue in Arcade was ");
         Put(Float(Arcade.Max_Wait), Fore => 4, Aft => 2, Exp => 0);
         New_Line;
      END Write_Final_Msg;

   end Display;
```

Finally, the main program randomises the random number generator, clears the screen, writes the headings, adds all the machines to the machine manager, then activates customer tasks with an arrival delay between each customer. When all the customers have been activated, wait until the arcade is empty, write the final message and then terminate

```
begin
   Reset(G);   -- randomise the random number generator
   Clearscreen;
   Put_Line("Machines customers  wait time");

   for I in 1..Number_Of_Machines loop
      Put_Line (Integer'Image(I));
   end loop;
   New_Line;
   Put_Line("Customers waiting");

   for I in 1..Number_Of_Machines loop
      Machine_Array(I).Your_Num_Is(I);  -- give Machine a counter position
      Machine_Mgr.Free(I);   -- and tell the manager the Machine is free
   end loop;

   for I in 1..Number_Of_Customers loop -- create customers
      Next_Customer := new Customer;    -- then wait between customer
      delay Duration(Ts*(Random(G) * Interarrival_Time));  -- arrivals
   end loop;
   -- now all created customers wait in the queue for a turn at a machine

   Arcade.IsEmpty;  -- wait till the arcade is empty

   Display.Write_Final_Msg;  -- farewell message with performance data

end Queue_manager;
```

And that is the end of the Queue_Manager simulation. It uses tasking to implement active objects for the customers and the machines and protected types to provide the support functions needed. The example includes an implementation of the bounded buffer, which can be used in many other applications.

17.11 A simplified gaming machine

This example makes good use of the select statement, including the delay timeout.

We want to design a game machine controller as a task that functions as follows.

a) The player starts with a small pool of funds.
b) The machine waits for the player to start.
c) The player makes a play and the machine debits the players funds on each play.
d) At random, the player may win a jackpot
e) When the funds run low, the player may add funds
f) If the machine is left idle, the player's funds are debited for idle time
g) If funds run out, the game terminates
h) The player may stop at any time and claim the winnings or remaining funds.

The machine task should be treated as an object with all it's data encapsulated and its actions triggered via messages sent to it via accept statements.

The actions of the game controller can be activated in any order since the user can start, play, add funds or quit at any time. This makes the design an ideal candidate for a multiway select statement using a guard and the list of functions above provide the pseudocode. We capture these requirements in a task, that I will call Bandit. Bandit starts with the usual preamble of "with"ing the packages it needs, declaring its entry points in its specification and its data types and variables. We also declare a random number generator, as we have done previously, to trigger jackpot wins at random. Here is the code:

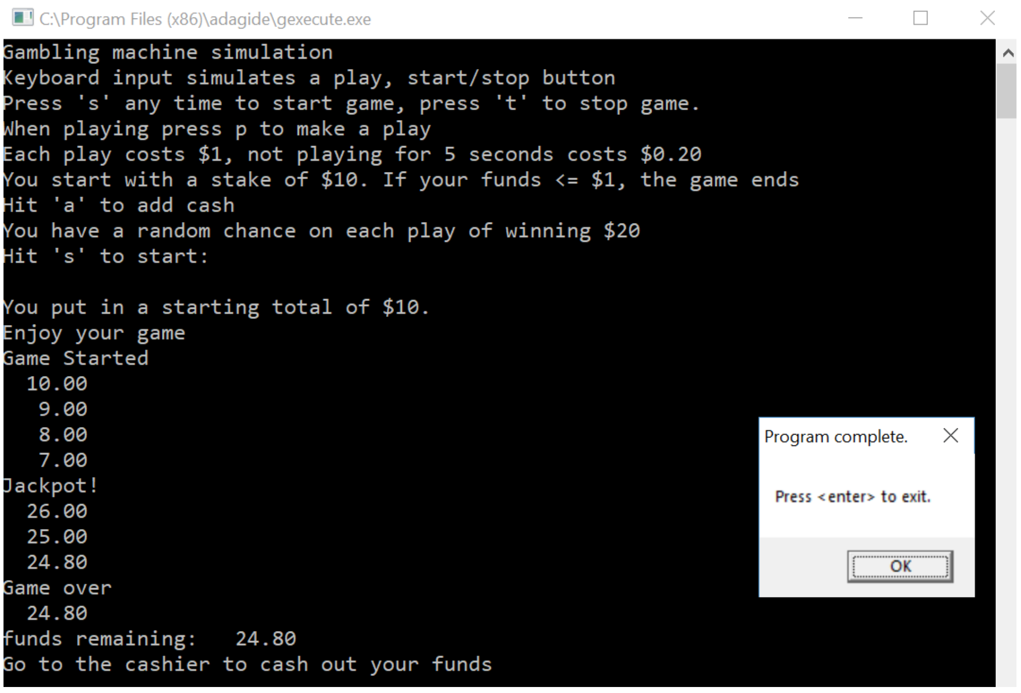
```
WITH Text_Io;
USE Text_Io;
WITH Ada.Numerics.Discrete_Random;

PROCEDURE GameControlV1 IS

   TASK Bandit IS
      ENTRY Start;
      ENTRY A_Play;
      entry Deposit;
      ENTRY Stop;
   END Bandit;

   TASK BODY Bandit IS
      TYPE Money IS DELTA 0.01 DIGITS 6 RANGE 0.0 .. 1000.0;
      PACKAGE Money_IO IS NEW Text_IO.Decimal_IO(Money);
      USE Money_IO;

      SUBTYPE Chance IS Integer RANGE 1..25;
      PACKAGE ChancePack IS NEW Ada.Numerics.Discrete_Random(Chance);
      USE ChancePack;  -- makes a random number in range 1..25
      G : Generator; -- for the random numbers

      Starting_Total : CONSTANT Money    := 10.00;    -- declare all variables
      Waiting_Charge : CONSTANT Money    := 0.10;
      Game_Charge    : CONSTANT Money    := 1.00;
      Wait_Interval  : CONSTANT Duration := 5.0;
      Cash           : Money;
      Total          : Money;
      Playing        : Boolean := False;
      Finished       : Boolean := False;
      Win            : Chance;
   BEGIN
      Reset(G);                                       -- Initialise random numbers
      WHILE NOT Finished LOOP
         SELECT
            WHEN NOT Playing =>
               ACCEPT Start;                    -- b) wait for player to start
            Put_Line("Game Started");
            Total := Starting_Total;            -- a) starting amount of cash
            Playing := True;
         OR
            ACCEPT Stop;                        -- h) can stop at any time
            Put_Line("Game over");              --    whether playing or not
            Finished := True;
         OR
            WHEN Playing =>
            ACCEPT A_Play;                      -- c)  The player makes a play
            Total := Total - Game_Charge;       --     and is debited
            Win := Random(G);
            IF Win = 12 THEN
               Total := Total + 20.00;          -- d) Wins jackpot
            END IF;
         OR
            WHEN Playing =>
            ACCEPT Deposit;                     -- e) add cash
            Put("Enter amount to add: ");
            Get(Cash);
            Total := Total + Cash;
         OR
            DELAY Wait_Interval;                -- f) timeout
            Total := Total - Waiting_Charge;
         END SELECT;
         Put(Total, Fore=>4, Aft=>2);           -- Show balance
         IF Total < Game_Charge THEN            -- g) Has cash run out?
            New_Line;
            Put_Line("Insufficient funds");
            Put_Line("Game over");
            Finished := True;
         END IF;
      END LOOP;                                 -- repeat until finished
      New_Line;
      Put("funds remaining: ");                 -- wrap up and farewell
      Put(Total, Fore=>4, Aft=>2);
      New_Line;
      Put_Line("Go to the cashier to cash out your funds");
   END Bandit;
```

As can be seen, this is a very neat implementation of the game machine. A real game machine would have push buttons and a simple display to generate the control signals, but we have to simulate this with some driver code to generate the control signals. We will ask the player to hit some keys to generate the signals:

```
   Next : Character;
BEGIN
   -- gambiling machibne driver
   Put_Line("Gambling machine simulation");
   Put_Line("Keyboard input simulates a play, start/stop button");
   Put_Line("Press 's' any time to start game, press 't' to stop game. ");
   Put_Line("When playing press p to make a play");
   Put_Line("Each play costs $1, not playing for 5 seconds costs $0.25");
   Put_Line("You start with a stake of $10. If your funds <= $1, the game ends");
   Put_Line("Hit 'a' to add cash");
   Put_Line("You have a random chance on each play of winning $20");
   Put_Line("Hit 's' to start: ");
   LOOP
      Get_Immediate(Next);
      IF Next = 's' THEN
         Bandit.Start;
      ELSIF Next = 't' THEN
```

```
            Bandit.Stop;
            EXIT;
         ELSIF Next = 'p' THEN
            Bandit.A_Play;
         ELSIF Next = 'a' THEN
            Bandit.Deposit;
         END IF;
         DELAY 0.01;  -- give Bandit time to terminate
         EXIT WHEN Bandit'Terminated;   -- then Main also terminates
      END LOOP;
   END GameControlV1;
```

Adding this driver code and running the resulting program produces output like this



```
Gambling machine simulation
Keyboard input simulates a play, start/stop button
Press 's' any time to start game, press 't' to stop game.
When playing press p to make a play
Each play costs $1, not playing for 5 seconds costs $0.20
You start with a stake of $10. If your funds <= $1, the game ends
Hit 'a' to add cash
You have a random chance on each play of winning $20
Hit 's' to start:

You put in a starting total of $10.
Enjoy your game
Game Started
  10.00
   9.00
   8.00
   7.00
Jackpot!
  26.00
  25.00
  24.80
Game over
  24.80
funds remaining:   24.80
Go to the cashier to cash out your funds
```

Program complete.

Press <enter> to exit.

This player luckily wins a jackpot after only 4 tries, tries once more, then waits a bit too long and is debited 20c before hitting 't' and ending.

The program appears to work well, until the player hits 'p' before first hitting 's', at which point the program locks and accepts no further input, just debiting the player 20c every 5 seconds. The program can only be terminated by letting the funds run out, closing the program execution window or hitting control-c. The problem is that while the task Bandit is still running, the input code has deadlocked.

17.12 Deadlock

Deadlock is a condition that arises in concurrent systems where a task makes a request for a resource or communication and the request cannot be satisfied because the resource is locked andcannot be unlocked, or the communication (rendezvous) cannot be accepted, causing the task to wait indefinitely, unable to proceed. There are many cases where deadlock can arise and a serious programmer should study the issue in one of the many books on concurrency, for example "Concurrency in Ada" by Burns and Wellings. In this case, if we hit 'p' before 's', the guard Running is false and the rendezvous at Bandit.A_Play cannot proceed, so the driver code stops at that point and no further input can be entered. The program is stuck!

We can take any of several approaches to the problem:

a) prevent the invalid input in the driver code,
b) accept all possible rendezvous combinations in the task Bandit and simply do nothing in Bandit when an invalid input occurs or
c) terminate a communication if it cannot proceed, using the ELSE clause as a SELECT alternative.

Solution a) would require breaking the encapsulation of task Bandit, so we discard it.

Solution b) would make the Select part of the code for Bandit look like this:

```
WHILE NOT Finished LOOP
   SELECT
      WHEN NOT Playing => ACCEPT Start;
      Put_Line("Game Started");
      Total := Starting_Total;
      Playing := True;
   OR
      WHEN NOT Playing => ACCEPT A_Play;  -- ** ignore these
   OR
      WHEN NOT Playing => ACCEPT Deposit; -- ** ie do nothing
   OR
      ACCEPT Stop;
      Put_Line("Game over");
      Finished := True;
   OR
      WHEN Playing =>
      ACCEPT A_Play;
      Win := Random(G);
      IF Win = 12 THEN
         Total := Total + Jackpot;
         Put("Jackpot, you win");
         Put_Line(Jackpot'Image);
      ELSE
         Total := Total - 1.00;
      END IF;
   OR
      WHEN Playing =>
      ACCEPT Start;  -- ** and do nothing - already playing
   OR
      WHEN Playing =>
      ACCEPT Deposit; -- deposit cash
      Put("Enter amount to add: ");
      Get(Cash);
      Total := Total + Cash;
   OR
      DELAY Wait_Interval;
      Total := Total - Waiting_Charge;
   END SELECT;
```

Note the 3 extra OR parts in the Select, marked with **, which ignore invalid communications. This is an example of defensive programming to guard against invalid inputs that can be the bane of a programmers life.

Solution c) adds a Select with an Else to the driver program communications to accept statements that have guards, so if a guard is up and the cornensponding rendezvous cannot be accepted, it is immediately abandoned so no lockout will occur. This requires a change to the driver code, as follows:

```
LOOP
   Get_Immediate(Next);
   IF Next = 's' THEN
      select
         Bandit.Start;
      ELSE
         NULL;
      END SELECT;
   ELSIF Next = 't' THEN
      Bandit.Stop;  -- stop is always accepted – does not need an ELSE
      EXIT;
   ELSIF Next = 'p' THEN
      SELECT
         Bandit.A_Play;
      ELSE
         NULL;
      END SELECT;
   ELSIF Next = 'a' THEN
      SELECT
         Bandit.Deposit;
      ELSE
         NULL;
      END SELECT;
   END IF;
   DELAY 0.01;  -- give Bandit time to terminate
   EXIT WHEN Bandit'Terminated;
END LOOP;
```

Both of these methods work in that they prevent the program from hanging up.

17.13 Avoiding Deadlock

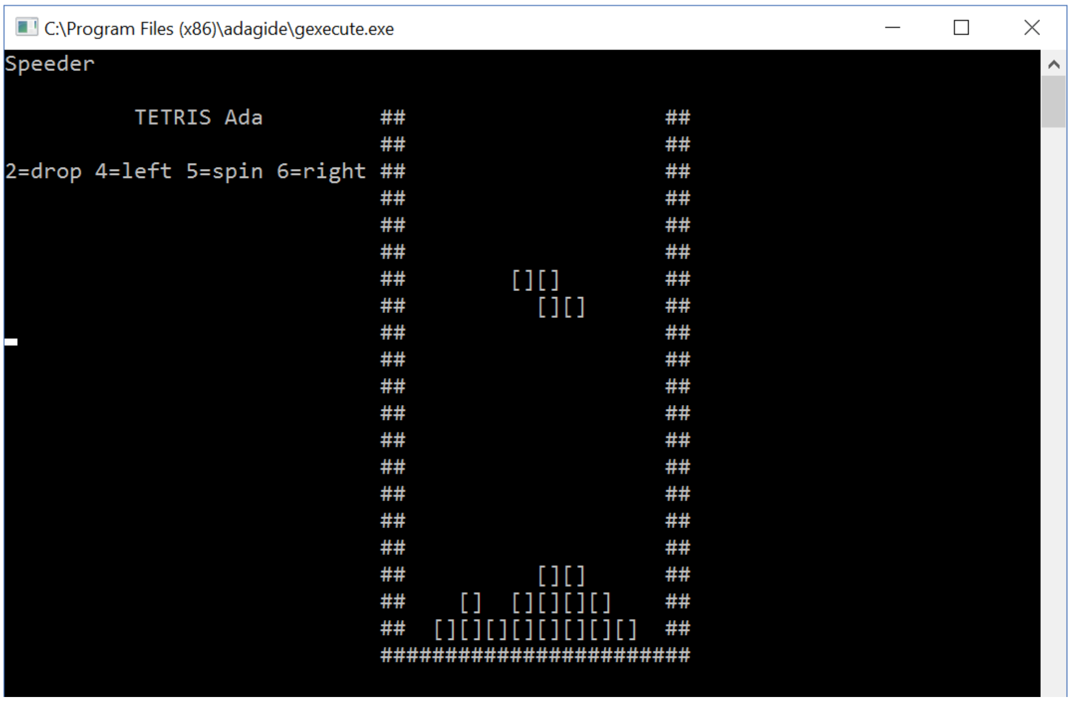
Avoiding deadlocks is a complex issue and a full treatment is beyond the scope of this short introduction to Ada. There are some fairly straightforward precautions a programmer can take, once aware of the potential problem. Here are a two.

a) When locking a resource (such as the Display, a printer or some data structure) always ansure it can be freed. If two or more resources must be locked, always lock them in the same order and free them in reverse order (eg lock A, lock B, free B, free A)
b) When using guards, always ensure a method exists to open the guard so any rendezvous locked against it can proceed. Alternatively, abandon any rendezvous that cannot immediately be accepted and take some alternative action.

17.14 Wrap up

For a final example of tasking and concurrency, I have included in the folder of code examples the game of tetris, tetris.adb written by Paul Pukite at the University of Minnesota, modified by me to run on windows using NT_Screen.

Tetris.adb running:


C:\Program Files (x86)\adagide\gexecute.exe

```
Speeder

          TETRIS Ada          ##                    ##
                             ##                    ##
2=drop 4=left 5=spin 6=right ##                    ##
                             ##                    ##
                             ##                    ##
                             ##                    ##
                             ##        [][]        ##
                             ##          [][]      ##
                             ##                    ##
                             ##                    ##
                             ##                    ##
                             ##                    ##
                             ##                    ##
                             ##                    ##
                             ##                    ##
                             ##                    ##
                             ##                    ##
                             ##          [][]      ##
                             ##    []  [][][][]    ##
                             ##  [][][][][][][][]  ##
                             ########################
```

Try running it and examine the code

This is the end of this introduction to Ada. There is a great deal more, since Ada provides many more facilities for real-time, other types of containers, advanced arithmetic, interfacing to other languages like C and C++, safety and security, etc. Look at the Annexes in the LRM for more.

Thank you for reading this book. Have fun!

Here are two good books on Ada for you to learn more:

John Barnes, “Programming in Ada 2012”

Norman Cohen, “Ada as A Second Language”

For much more information, tutorials, books and examples, go to the Ada Informations Clearing House at http://www.adaic.org

David Levy, © 2018, with thanks to Michael Dawson, Mike Kamrad, Paul Pukite and E Burke