Ada–A Crash Course

Peter Chapin*
Vermont Technical College

Generated: July 27, 2015

*PChapin@vtc.vsc.edu

Contents

1 Tutorial

Welcome to the Ada programming language! The purpose of this tutorial is to give you an overview of Ada so that you can start writing Ada programs quickly. This tutorial does not attempt to cover the entire language. Ada is very large, so complete coverage of all its features would take many more pages than are contained in this document. However, it is my hope that after reading this tutorial you will have a good sense of what Ada is like, appreciate some of its nicer features, and feel interested in learning more [**?**, **?**, **?**, **?**, **?**].

This tutorial assumes you are already familiar with one or more languages in the C family: C, C++, Java, C#, or something similar. It is not my intention to teach you how to program. I assume you already understand concepts such as loops, data types, functions, and so forth. Instead this tutorial describes how to use these things in Ada. In cases where Ada provides features that might be unfamiliar to you (such as subtypes, discriminated types, and tasking) I will discuss those features a bit more comprehensively.

Ada is a powerful language designed to address the following issues:

- The development of very large programs by multiple, loosely connected teams. The language has features to help manage a large number of program components, and to help ensure those components are used consistently.
- The development of long lived programs that spend most of their time in the maintenance phase of the software life cycle. The language is designed to promote the readability of programs. You may find Ada code to be rather verbose and tedious to write. However, that extra work pays off later by making the code clearer and easier to read when bugs must be fixed or enhancements written.
- The development of robust programs where correctness, security, and reliability are priorities. The language has features designed to make programming safer and less error prone. Some of these features involve extra run time checking and thus entail a performance penalty. However, Ada's design is such that the performance penalty is normally not excessive.
- The development of embedded systems where low level hardware control, multiple concurrent tasks, and real time requirements are common. The language has features designed to support these things while still retaining as much safety as feasible.

1.1 Hello, Ada

Whenever you begin to study a new language or software development environment you should start by writing the most trivial program possible in that language or environment. Thus I will begin this tutorial with a short but complete program that displays the string "Hello, Ada!" on the console.

```
with Ada.Text_IO;

procedure Hello is
begin
   Ada.Text_IO.Put_Line("Hello, Ada!");
end Hello;
```

The program starts with a *context clause* consisting of a **with** statement. The context clause specifies the packages that will be used in this particular compilation unit. Although the compilation unit above contains just a single procedure, most Ada code is in packages. The standard library components are in child packages of the package Ada. In this case we will be using facilities in the child package Ada.Text_IO.

The main program is the procedure named Hello. The precise name used for the main program can be anything; exactly which procedure is used as the program's entry point is specified when the program is compiled. The procedure consists of two parts: the part between **is** and **begin** is called the declarative part. Although empty in this simple case, it is here where you would declare local variables and other similar things. The part between **begin** and **end** constitute the executable statements of the procedure. In this case, the only executable statement is a call to procedure Put_Line in package Ada.Text_IO. As you can probably guess from its name, Put_Line prints the given string onto the program's standard output device. It also terminates the output with an appropriate end-of-line marker.

Notice how the name of the procedure is repeated at the end. This is optional, but considered good practice. In more complex examples the readability is improved by making it clear exactly what is ending. Notice also the semicolon at the end of the procedure definition. C family languages do not require a semicolon here so you might accidentally leave it out.

Spelling out the full name Ada.Text_IO.Put_Line is rather tedious. If you wish to avoid it you can include a **use** statement for the Ada.Text_IO package in the context clause (or in the declarative part of Hello). Where the **with** statement makes the names in the withed package *visible*, the **use** statement makes them *directly visible*. Such names can then be used without qualification as shown below:

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

procedure Hello is
begin
   Put_Line("Hello, Ada!");
end Hello;
```

Many Ada programmers do this to avoid having to write long package names all the time. However, indiscriminate use of **use** can make it difficult to understand your program because it can be hard to tell in which package a particular name has been declared.

Ada is a case insensitive language. Thus identifiers such as Hello, hello, and hElLo all refer to the same entity. It is traditional in modern Ada source code to use all lower case letters for reserved words. For all other identifiers, capitalize the first letter of each word and separate multiple words with an underscore.[1] The Ada language does not enforce this convention but it is a well established standard in the Ada community so you should follow it.

Before continuing I should describe how to compile the simple program above. I will assume you are using the freely available GNAT compiler. This is important because GNAT requires specific file naming conventions that you must follow. These conventions are not part of the Ada language and are not necessarily used by other Ada compilers. However, GNAT depends on these conventions in order to locate the files containing the various compilation units of your program.

The procedure Hello should be stored in a file named `hello.adb`. Notice that the name of the file must be in lower case letters and must agree with the name of the procedure stored in that file. The `adb` extension stands for "Ada body." This is in contrast with Ada specification files that are given an extension of `ads`. You will see Ada specifications when I talk about packages in Section **??**. I will describe other GNAT file naming requirements at that time.

To compile `hello.adb`, open a console (or terminal) window and use the gnatmake command as follows

```
> gnatmake hello.adb
```

The `gnatmake` command will compile the given file and link the resulting object code into an executable producing, in the above example, `hello.exe` (on Windows). You can compile Ada programs without `gnatmake` by running the compiler and linker separately. There are sometimes good reasons to do that. However, for the programs you will write as a beginning Ada programmer, you should get into the habit of using `gnatmake`.

Note that GNAT comes with a graphical Ada programming environment named GPS (GNAT Programming Studio). GPS is similar to other modern integrated development environments such as Microsoft's Visual Studio or Eclipse. Feel free to experiment with GPS if you are interested. However, the use of GPS is outside the scope of this tutorial.

When the compilation has completed successfully you will find that several additional files have been created. The files `hello.o` and `hello.exe` (on Windows) are the object file and executable file respectively. The file `hello.ali` is the Ada library information file. This file is used by GNAT to implement some of the consistency checking required by the Ada language. It is a plain text file; feel free to look at it. However, you would normally ignore the `ali` files. If you delete them, they will simply be regenerated the next time you compile your program.

[1] This style is often called "title case."

Exercises

1. Enter the trivial "Hello, Ada" program on page **??** into your system. Compile and run it.

2. Make a minor modification to the trivial program that results in an error. Try compiling the program again. Try several different minor errors. This will give you a feeling for the kinds of error messages GNAT produces.

3. Experiment with the **use** statement. Try calling Put_Line without specifying its package, both with and without the **use** statement. Try putting the **use** statement inside the declarative part of the procedure. Try putting the **with** statement inside the declarative part of the procedure.

1.2 Control Structures

Ada contains all the usual control structures you would expect in a modern language. The program in Listing **??** illustrates a few of them, along with several other features. This program accepts an integer from the user and checks to see if it is a prime number.

Listing 1.1: Prime Checking Program

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

procedure Prime is
   Number : Integer;
begin
   Put("Enter an integer: ");
   Get(Number);
   if Number < 2 then
      Put("The value "); Put(Number, 0); Put_Line(" is bad.");
   else
      Put("The value "); Put(Number, 0);
      for I in 2 .. (Number - 1) loop
         if Number rem I = 0 then
            Put_Line(" is not prime.");
            return;
         end if;
      end loop;
      Put_Line(" is prime.");
   end if;
end Prime;
```

The program declares a local variable named Number of type Integer in its declarative part. Notice that the type appears after the name being declared (the opposite of C family languages), separated from that name with a colon.

Procedure Get from package Ada.Integer_Text_IO is used to read an integer from the console. If the value entered is less than two an error message is displayed. Notice that there are two different Put procedures being used: one that outputs a string and another that outputs an integer. Like C++, Java, and some other modern languages, Ada allows procedure names to be overloaded distinguishing one procedure from another based on the parameters. In this case the two different Put procedures are also in different packages but that fact is not immediately evident because of the **use** statements.

If the value entered is two or more, the program uses a **for** loop to see if any value less than the given number divides into it evenly (that is, produces a zero remainder after division, as calculated with the **rem** operator). Ada **for** loops scan over the given range assigning each value in that range to the loop parameter variable (named I in this case) one at a time. Notice that it is not necessary to explicitly declare the loop parameter variable. The compiler deduces its type based on the type used to define the loop's range. It is also important to understand that if the start of the range is greater than the end of the range, the result is an empty range. For example, the range 2 .. 1 contains no members and a loop using that range won't execute at all. It does not execute starting at two and counting down to one. You need to use the word **reverse** to get that effect:

```
for I in reverse 1 .. 2 loop
```

Notice also that **if** statements require the word **then** and that each **end** is decorated by the name of the control structure that is ending. Finally notice that a single equal sign is used to test for equality. There is no "==" operator in Ada.

The program in Listing **??** counts the vowels in the text at its standard input. You can provide data to this program either by typing text at the console where you run it or by using your operating system's I/O redirection operators to connect the program's input to a text file.

This program illustrates **while** loops and **case** structures (similar to C's switch statement). There are quite a few things to point out about this program. Let's look at them in turn.

- Variables can be initialized when they are declared. The := symbol is used to give a variable its initial value (and also to assign a new value to a variable). Thus like C family languages, Ada distinguishes between test for equality (using =) and assignment (using :=). Unlike C family languages you'll never get them mixed up because the compiler will catch all incorrect usage as an error.

- In this program the condition in the **while** loop involves calling the function End_Of_File in package Ada.Text_IO. This function returns True if the standard input device is in an end-of-file state. Notice that Ada does not require (or even allow) you to use an empty parameter list on functions that take no parameters. This means that function End_Of_File looks like a variable when it is used. This is considered a feature; it means that read-only variables can be replaced by parameterless functions without requiring any modification of the source code that uses that variable.

- The program uses the logical operator **not**. There are also operators **and** and **or** that can be used in the expected way.

Listing 1.2: Vowel Counting Program

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

procedure Vowels is
   Letter       : Character;
   Vowel_Count : Integer := 0;
   Y_Count      : Integer := 0;
begin
   while not End_Of_File loop
      Get(Letter);
      case Letter is
         when 'A'|'E'|'I'|'O'|'U' |
              'a'|'e'|'i'|'o'|'u' =>
            Vowel_Count := Vowel_Count + 1;

         when 'Y'|'y' =>
            Y_Count := Y_Count + 1;

         when others =>
            null;
      end case;
   end loop;
   Put("Total number of vowels = "); Put(Vowel_Count); New_Line;
   Put("Total number of Ys = "); Put(Y_Count); New_Line;
end Vowels;
```

- The **case** statement branches to the appropriate **when** clause depending on the value in the variable Letter. The two **when** clauses in this program show multiple alternatives separated by vertical bars. If any of the alternatives match, that clause is executed. You can also specify ranges in a **when** clause such as, for example, **when** 1 .. 10 => etc.

- Unlike C family languages there is no "break" at the end of each **when** clause. The flow of control does *not* fall through to the next **when** clause.

- Ada has what are called *full coverage rules*. In a **case** statement you must account for every possible value that might occur. Providing **when** clauses for just the vowels would be an error since you would not have fully covered all possible characters (the type of Letter is Character). In this program I want to ignore the other characters. To do this, I provide a **when others** clause that executes the special **null** statement. This lets future readers of my program know that I am ignoring the other characters intentionally and it's not just an oversight.

- New_Line is a parameterless procedure in package Ada.Text_IO that advances the output to the next line. More accurately, New_Line is a procedure with a default parameter that can be used to specify the number of lines to advance, for example: New_Line(2).

Exercises

1. The prime number program in Listing **??** is not very efficient. It can be improved by taking advantage of the following observation: If Number is not divisible by any value less than I, then it can't be divisible by any value greater than Number/I. Thus the upper bound of the loop can be reduced as I increases. Modify the program to use this observation to improve its performance. Note: You will have to change the **for** loop to a **while** loop (try using a **for** loop first and see what happens).

2. Modify the prime number program again so that it loops repeatedly accepting numbers from the user until the user types -1. You can program an infinite loop in Ada as shown below. Have your program print different error messages for negative numbers (other than -1) than for the values zero and one. Use an **if** ... **elsif** chain (note the spelling of **elsif**). Write a version that uses a **case** statement instead of an **if** ... **elsif** chain. You can use Integer'First in a range definition to represent the first (smallest) Integer value the compiler can represent.

```
loop
   ...
   exit when condition
   ...
end loop;
```

3. *Challenging.* The vowel counting program in Listing **??** counts 'Y' separately because in English 'Y' is sometimes a vowel and sometimes not. Modify the program so that it adds occurrences of 'Y' to the vowel counter only when appropriate.

1.3 Types and Subtypes

In the examples so far you have seen the built-in types Integer and Character. Ada also has a type Float for floating point values and a type Boolean for true/false values. These types are similar to their counterparts in other programming languages. However, it may surprise you to learn that, with the possible exception of Boolean, these built-in types are often not used directly. Instead Ada encourages you to define types that reflect your problem domain and then use your types throughout your program. There is nothing new about this idea; many languages support some mechanism for creating user defined types. However, in Ada it is possible to create even user defined integer, floating point, and character types—a feature many languages do not support.

To explore this idea further, I will start by introducing Ada's *strong typing*. There is no precise definition of this term, but for our purposes we will say that a strongly typed language is one which does very few, if any, implicit type conversions. Unlike C, for example, Ada will not convert integers to floating point values or vice-versa without explicit instruction from the programmer. The example procedure in Listing **??** illustrates the effect.

Listing 1.3: Vowel Counting Program

```
procedure Strong_Typing_Example is
   I : Integer;
   F : Float;
begin
   I := 1;     -- Okay.
   I := 1.0;   -- Error. Can't assign a Float to an Integer.
   F := 1;     -- Error. Can't assign an Integer to a Float.
   F := 1.0;   -- Okay.
   F := I;     -- Error.
   I := F;     -- Error.
   F := Float(I);     -- Okay. Explicit conversion.
   I := Integer(F);   -- Okay. Explicit conversion.
end Strong_Typing_Example;
```

If you are not used to strong typing you may find it excessively pedantic and annoying. However, it exists for a reason. Experience has shown that many bugs first manifest themselves as confusion about types. If you find yourself mixing types in your program, it may mean that you have a logical error. Does it make sense to put a value representing aircraft velocity into a variable that holds a passenger count? Strong typing can catch errors like this, and thus it promotes software reliability.

By creating a new type for each logically distinct set of values in your program, you enable the compiler to find logical errors that it might otherwise miss. For example, suppose you are working on a program that manipulates a two dimensional array of graphical pixels. Instead of using Integer to represent pixel coordinates you might define separate types as follows:

```
type X_Coordinate_Type is new Integer;
type Y_Coordinate_Type is new Integer;
```

Here the two types X_Coordinate_Type and Y_Coordinate_Type are distinct types with no implicit conversion from one to the other, even though fundamentally both are integer types. Yet because they are distinct, assigning a Y coordinate value to a variable intended to hold an X coordinate is now a compile-time error rather than a mistake to be found, if you are lucky, during testing. For example suppose you had the variables:

```
Base_Length : X_Coordinate_Type;
Side_Length : Y_Coordinate_Type;
```

Now an assignment such as:

```
Side_Length := Base_Length;
```

is caught by the compiler as a type mismatch error. If you actually desire to do this assignment anyway because, perhaps, you are talking about a square figure, you can use an explicit type conversion:

```
Side_Length := Y_Coordinate_Type(Base_Length);
```

You might feel that distinguishing between coordinate values in two dimensions is overly restrictive. Yet it may still make sense to define a separate Coordinate_Type to distinguish values that are pixel coordinates from other kinds of integers in your program such as, for example, line counters, TCP/IP port numbers, user ID numbers, and so forth. In a language like C where a single type "int" might be used to hold all the values I mentioned, the compiler is powerless to detect illogical mixing of those values across what are really totally different conceptual domains.

In C++ one could create a class representing each logical concept, and then use overloaded operators to make instances of that class appear integer-like. However, that is a very heavy-handed approach to solve what should be a simple problem. Thus, it is not commonly done.[2]

Ada also allows you to constrain the range of allowed values when you define a new scalar type. Consider the pixel coordinate example again, and suppose that the drawing area is limited to 2048 pixels in width and 1024 pixels in height. These limitations can be made known to Ada's type system using type definitions such as:

```
type X_Coordinate_Type is range 0 .. (2048 − 1);
type Y_Coordinate_Type is range 0 .. (1024 − 1);
```

[2]If you are a C++ person, as an exercise try writing a C++ template that allows you to "easily" define distinct, integer-like types. Compare your result with the Ada facility described here.

Here the (still distinct) types are defined as constrained ranges on Integer with the bounds on X_Coordinate_Type running from 0 to 2047 inclusive. When defining new types like this Ada requires that the ranges be *static expressions* that can be computed by the compiler. Thus, an expression such as 2048 − 1 is acceptable but an expression involving a variable is not.

Whenever a value is assigned to a variable the compiler inserts runtime checks, if necessary, to verify that the assigned value is in the range of that variable's type. If a range check fails, the Constraint_Error exception is raised. Notice that, unlike the type checking, range checks are done at runtime. For example:

```
Side_Length := Y_Coordinate_Type(Base_Length); -- Might raise Constraint_Error.
Base_Length := X_Coordinate_Type(Side_Length); -- No Constraint_Error possible.
Side_Length := 1024;  -- Will raise Constraint_Error at runtime.
Side_Length := 1023;  -- No Constraint_Error possible.
```

The first assignment above might raise Constraint_Error because the value in Base_Length could be out of range for type Y_Coordinate_Type. However, the compiler can't know for sure (in general) without running your program, and so the check is done at runtime. However, the compiler can be sure that the second assignment above will not raise Constraint_Error since every possible value of Y_Coordinate_Type is legitimate for X_Coordinate_Type. Thus the compiler is encouraged, although not required, to "optimize away" the runtime check.

The third assignment above clearly tries to put an out of range value into Side_Length but the program will compile anyway. Failed range checks are uniformly handled at runtime; the Ada language does not require compilers to detect any of them at compile time, no matter how obvious they may be. That said, a reasonable compiler will notice the obvious range violation in this example and produce a warning message about it.

Defining your own types also helps you write portable programs. The only integer type compilers must support is Integer. The number of bits used by type Integer is implementation defined and varies from compiler to compiler. Most compilers support additional, non-standard integer types such as Long_Integer and Long_Long_Integer. However, the names of these additional types, and even their existence, varies from compiler to compiler. Rather than deal with this complexity directly you can just specify the range of values you desire and the compiler will select the most appropriate underlying type. For example

```
type Block_Counter_Type is range 0 .. 1_000_000_000;
```

Variables of type Block_Counter_Type may be represented as Integer or Long_Integer or some other type as appropriate. In any case they will be constrained to only hold values between zero and one billion inclusive. If the compiler does not have a built-in integer type with a large enough range it will produce an error message. If the compiler accepts your type definition you will not get any surprises. Notice also how Ada allows you to embed underscores in large numbers to improve their readability.

Ada also allows you to define *modular types*. These types are unsigned and have "wrap-around" semantics. Incrementing beyond the end of an ordinary type causes an exception, but incrementing beyond the end of a modular type wraps around to zero. In

addition the operators **not**, **and**, **or**, and **xor** can be used on modular types to do bitwise manipulation. The following listing demonstrates this:

```
type Offset_Type is mod 2**12; -- Two to the 12th power.
-- The range of Offset_Type is 0 .. 2**12 - 1.
...
Offset : Offset_Type;
...
Offset := Offset and 16#3FF#; -- Bitwise AND with a hex mask.
```

Here a modular type Offset_Type is introduced to hold 12 bit offsets. The variable Offset is masked so that only the lower 10 bits are retained. The snippet above also demonstrates Ada's exponentiation operator and how numbers in bases other than 10 can be written. Because Ada was designed for use in embedded systems, its support for low level bit manipulation is good.

1.3.1 Enumeration Types

In many cases you want to define a type that has only a small number of allowed values and you want to name those values abstractly. For example, the definition

```
type Day_Type is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
```

introduces Day_Type as an *enumeration type*. Variables of type Day_Type can only take on the values listed in the type definition. Those values are called *enumerators*. This is useful for writing your program in abstract terms that are easy to understand. Because the compiler treats enumeration types as distinct from integer types (and from each other) you will not be allowed to mix unrelated concepts without compile time errors.

Enumeration types are useful for modeling real world entities that have a limited number of distinct states. They can also be used as more descriptive alternatives to the built-in type Boolean. For example:

```
type Color_Type is (Red, Yellow, Green);
type State_Type is ( Initializing , Waiting, Processing, Stopping);

-- These types are alternatives to Boolean's True/False.
type Switch_Setting is (Off, On);
type Control_Rod_Setting is (Down, Up);
type System_Mode is (Sleeping, Active);
```

Actually, the built-in type Boolean is an enumeration type; it behaves exactly as if it was defined as:

```
type Boolean is (False, True);
```

Even the built-in type Character is an enumeration type where the enumerators are the character literals.

1.3.2 Discrete Types

The integer types and enumeration types are *discrete types* because they each represent only a finite set of (ordered) values. All discrete types share some common properties and it is enlightening to describe those properties in a uniform way.

Ada makes extensive use of *attributes*. You reference an attribute of a type using an apostrophe (or "tick") immediately following the type name. For example, Day_Type'First represents the first value in the Day_Type enumeration and Integer'First represents the first (smallest) allowed value of Integer. There is also a Last attribute to access the last allowed value. Some attributes behave like functions. For example Day_Type'Succ(Sun) returns Mon, the successor of Sun in the Day_Type enumeration. There is also a Pred attribute for finding the predecessor value.

In addition there is a Pos attribute that returns the integer position of a particular enumerator and a Val attribute that does the reverse. For example Day_Type'Pos(Mon) is one (the position values start at zero) and Day_Type'Val(1) is Mon. Ada defines many attributes and we will see others later in this tutorial.

Ranges of discrete types can be defined in a uniform way and used, for example, as cases in **case** statements and ranges of **for** loops. The following listing shows a contrived example:

```
for Day in Sun .. Fri loop
  case Day in
    when Sun       => -- etc...
    when Mon | Tue => -- etc...
    when Wed .. Fri => -- etc...
  end case;
end loop;
```

Notice that despite Ada's full coverage rules it is not necessary to specify a case for Sat. This is because the compiler can see Day is limited to Sun .. Fri and thus will never take on the value Sat. Recall that the loop index variable of a **for** loop can not be modified inside the loop so there is no possibility of the programmer setting Day to Sat before the **case** statement is reached.

1.3.3 Subtypes

As we have seen, when a new type is defined using **type**, the compiler regards it as distinct from all other types. This is strong typing. However, sometimes it is better to define a constraint on an existing type rather than introduce an entirely new type. There are several different kinds of constraints one might define depending on the type being constrained. In this section I will talk about range constraints on discrete types. As an example, consider the definition:

```
subtype Weekday is Day_Type range Mon .. Fri;
```

This introduces a *subtype* named Weekday that only contains the values Mon through Fri. It is important to understand that a subtype is not a new type, but just a name for a constrained version of the parent type. The compiler allows variables of a subtype to be

mixed freely with variables of the parent type. Runtime checks are added if necessary to verify that no value is ever stored in a variable outside the range of its subtype. If an attempt is made to do so, the Constraint_Error exception is raised. The following listing shows an example:

```
procedure Demonstrate_Subtypes
  (Lower, Upper : in Day_Type; Day : in out Day_Type) is

  subtype Interval is Day_Type range Lower .. Upper;
  X : Interval := Interval'First;
begin
  Day := X; -- No run time check. Will definitely succeed.
  X := Day; -- Run time check. Day might be out of range.
End Demonstrate_Subtypes;
```

In this example a subtype Interval is defined in the procedure's declarative part. The variable X of type Interval is given Interval's first value. Mixing Interval and Day_Type variables in the later assignment statements is allowed because they are really both of the same type. Because Interval is a subtype of Day_Type the assignment of X to Day must succeed. Thus the compiler does not need to include any runtime checks on the value of X. However, the value of Day might be outside the allowed range of the subtype and so the compiler will need to insert a runtime check on Day's value before assigning it to X. If that check fails, the Constraint_Error exception is raised. Under no circumstances can an out of bounds value be stored in a variable.

I should point out that in this particular (simplistic) example the compiler's optimizer may be able to see that the value in Day will be in bounds of the subtype since in the assignment just before it was given a value from the subtype. In that case the compiler is allowed, and encouraged, to optimize away the run time check that would normally be required. In general Ada allows any check to be optimized away if the compiler can prove the check will never fail since doing so will cause no observable change to the program's behavior.

This example also illustrates another important aspect of subtypes: unlike full type, subtypes can be dynamically defined. Notice that the range on the subtype is taken from the procedure's parameters. Each time the procedure is called that range might be different. In contrast, as mentioned previously, the range specified on full type definitions must be static.

A range such as 1 .. 10 is really an abbreviation for the specification of a subtype:

```
Integer range 1 .. 10
```

Thus a **for** loop header such as **for** I **in** 1 .. 10 is really just an abbreviation for the more specific **for** I **in** Integer **range** 1 .. 10. In general the **for** loop specifies a subtype over which the loop index variable ranges. This is why above it was not necessary to provide a case for Sat. The loop parameter Day implicitly declared in the loop header has the subtype Day_Type **range** Sun .. Fri. The **case** statement contains alternatives for all possible values in that subtype so the full coverage rules were satisfied.

The Ada environment predefines two important and useful subtypes of Integer. Although you never have to explicitly define these types yourself, the compiler behaves as

if the following two subtype definitions were always directly visible:

```
subtype Natural is Integer range 0 .. Integer'Last;
subtype Positive is Integer range 1 .. Integer'Last;
```

The subtype Natural is often useful for counters of various kinds. Since counts can't be negative the run time checking on Natural's range constraint can help you find program errors. The subtype Positive is useful for cases where zero is a nonsensical value that should never arise. It is also often used for array indexes as you will see.

Exercises

1. Using the definition of Day_Type presented earlier would you guess that the following works or produces a compiler error: **for** Day **in** Day_Type **loop** ...? Write a short program to see how the compiler behaves when presented with a loop like this.

2. In the discussion above separate X_Coordinate_Type and Y_Coordinate_Type definitions where introduced. What disadvantage is there to keeping these concepts distinct? Would it be better to use a single Coordinate_Type for both X and Y coordinates instead? Discuss the advantages and disadvantages of both approaches.

3. One question that often comes up in the design of Ada programs is if a certain concept should get a distinct type or be defined as a subtype of some other type. What are the relative advantages and disadvantages of full types versus subtypes? Think of an example (other than the ones discussed here) where two concepts are best represented as separate types, and an example where two concepts are best represented by one being a subtype of the other.

4. Pick a program you've written in some other language and identify the different ways in which the integer type is used. If you were to translate your program to Ada, which of those ways would best get new types and which would best be subtypes? Show appropriate type/subtype definitions.

1.4 Subprograms

Unlike C family languages, Ada distinguishes between procedures and functions. Specifically, functions must return a value and must be called as part of a larger expression. Procedures never return a value (in the sense that functions do) and must be called in their own statements. Collectively procedures and functions are called subprograms in situations where their differences don't matter.

Subprograms can be defined in any declarative part. Thus it is permissible to nest subprogram definitions inside each other. A nested subprogram has access to the parameters and local variables in the enclosing subprogram that are defined above the nested subprogram. The scope rules are largely what you would expect. Listing **??** shows a variation of the prime number checking program first introduced on page **??**. This variation introduces a nested function Is_Prime that returns True if its argument is a prime number.

Listing 1.4: Prime Checking Program, Version 2

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

procedure Prime2 is
   Number : Positive;

   -- This function returns True if N is prime; False otherwise.
   function Is_Prime(N : Positive) return Boolean is
   begin
      for I in 2 .. (N      1) loop
         if N mod I = 0 then
            return False;
         end if;
      end loop;
      return True;
   end Is_Prime;

begin
   Put("Enter an positive integer: ");
   Get(Number);
   if Number < 2 then
      Put("The value "); Put(Number, 0); Put_Line(" is bad.");
   else
      Put("The value "); Put(Number, 0);
      if Is_Prime(Number) then
         Put_Line(" is prime.");
      else
         Put_Line(" is not prime.");
      end if;
   end if;
end Prime2;
```

In this case Is_Prime has been declared to take a parameter N of type Positive. It could have also accessed the local variable Number directly. However, it is usually better style to pass parameters to subprograms where it makes sense. In this case I also wanted to illustrate the syntax for parameter declarations. Note that because Is_Prime is a function, a return type must be specified and it must be called in the context of an expression, such as in the condition of an **if** statement as is done in Listing **??**.

Procedure declarations are similar except that no return type is mentioned. Also when a procedure is called it must not be part of an expression, but rather stand by itself as a single statement. See the calls to procedures Put and Put_Line in the examples so far.

The program above also shows the form of a comment. Ada comments start with −− and run to the end of the line. As you know, it is good to include comments in your program. However, the examples in this tutorial do not include many comments in order to save space and because the programs are explained in the text anyway.

Subprogram parameters can also have one of three possible "modes." The listing below shows the definition of a procedure that illustrates this. Notice that a semicolon is used to separate parameter declarations and not a comma as is done in C family languages.

```
procedure Mode_Example
  (X : in Integer; Y : out Integer; Z : in out Integer) is
begin
  X := 1;      -- Error. Can't modify an in parameter.
  Y := X;      -- Okay. Can read X and modify Y.
  Z := Z + 1;  -- Okay. Can read and write an in out parameter.
end Mode_Example;
```

The first parameter declared above has mode **in**. Parameters with this mode are initialized with the argument provided by the caller, but treated as constants inside the procedure. They can be read, but not modified.

The second parameter has mode **out**. Parameters with this mode are effectively in an uninitialized state when the procedure begins, but they can be used otherwise as ordinary variables inside the procedure. In particular **out** parameters can be read provided you first write a value into them. Whatever value is assigned to an **out** parameter by the time the procedure ends is sent back to the calling environment.

The third parameter has mode **in out**. Parameters with this mode are initialized with the argument provided by the caller, and can also be modified inside the procedure. The changed value is then returned to the caller to replace the original value. Keep in mind that, unlike in C, modifications to parameters in Ada (when allowed by the parameter mode) affect the arguments used when the procedure is called.

The mode **in** is the default. In versions of Ada prior to Ada 2012, functions parameters could also only have mode **in**, and thus it is common to leave the mode specification off when defining function parameters. Modern Ada allows function parameters with any of the three modes. My recommendation, however, is to always specify the mode when declaring procedure parameters but accept the default of **in**, except under special circumstances, when declaring function parameters. Functions with parameters of mode **out** or **in out** should be rare. Reasoning about a program where functions modify their

arguments can be difficult.

Like C++, Ada also allows you to define default values for subprogram parameters. This is accomplished by initializing the parameter (using the := assignment symbol) when the parameter is declared. If no argument is provided for that parameter when the subprogram is called the default value is used instead.

Ada also allows you to call subprograms using *named parameter associations*. This method allows you to associate arguments with the parameters in any order, and it can also greatly improve the readability of your code—particularly if you've chosen good names for the parameters. The listing below shows how the procedure defined above might be called. Assume that the variables Accumulator, N, and Output have been previously declared as integers.

```
Mode_Example(Z => Accumulator, X => N+15, Y => Output);
```

Notice that the order in which the arguments are provided is not the same as the order in which the parameters are declared. When named association is used, the order is no longer significant. Notice also that you can use any expression as the argument associated with an **in** parameter. However, **out** and **in out** parameters must be associated with variables and not arbitrary expressions since putting a value into the result of an expression doesn't really make sense. The result of an expression is an anonymous temporary variable, and there is no way for you to access its value.

Exercises

1. Write a procedure Count_Primes that accepts a range of positive integers and returns the number of primes, the smallest prime, and the largest prime in that range. Your implementation should use **out** parameters to return its results. It should also make use of the Is_Prime function defined earlier. Wrap your procedure in a main program that accepts input values from the user and outputs the results. Use named parameter association when calling Count_Primes.

2. Implement Count_Primes from the previous question as a function (or a collection of functions) instead. What are the advantages and disadvantages of each approach? After reading the following section on records in Ada, implement Count_Primes to return a record containing the three result values. What are the advantages and disadvantages of this approach?

1.5 Arrays and Records

Unlike in C, arrays in Ada are first class objects. This means you can assign one array to another, pass arrays into subprograms and return arrays from functions. Every array has a type. However, the type of an array does not need to be named. For example:

```
Workspace : array(1 .. 1024) of Character;
```

defines Workspace to be an array of characters indexed using the integers 1 through 1024. Note that although Workspace has an array type, the name of that type is not specified.

In fact, you can define arrays using any discrete subtype for the index Recall that 1 .. 1024 is really an abbreviation for an integer subtype specification. It is not possible to access an array out of bounds for exactly the same reason it's not possible to store an out of bounds value into a variable. The compiler raises the Constraint_Error exception in precisely the same way.

The following example:

```
procedure Example is
  type Day_Type is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
  Work_Hours : array(Day_Type) of Natural;

  function Adjust_Overtime
    (Day : Day_Type; Hours : Natural) return Natural is
  begin
    -- Not shown.
  end Adjust_Overtime;

begin
  Work_Hours := (0, 8, 8, 8, 8, 0);
  for Day in Day_Type loop
    Work_Hours(Day) := Adjust_Overtime(Day, Work_Hours(Day));
  end loop;
end Example;
```

This example illustrates several important features.

1. Arrays are accessed using parenthesis and not square brackets as in C family languages. Thus array access has a syntax similar to that of a function call. An array of constant values and can be replaced with a function later without clients needing to be edited.

2. Any discrete subtype, including enumeration subtypes, can be used for an array index. In the preceding example the array Work_Hours has elements Work_Hours(Sun) and so forth.

3. The nested function Adjust_Overtime uses a type defined in the enclosing procedure. This is perfectly acceptable. The visibility of all names is handled uniformly.

4. It is possible to use an *array aggregate* in a expression where an array is expected. Work_Hours is assigned a value that specifies every element of the array in one step. The compiler deduces the type of the aggregate from its context.

5. Notice that in the preceding example the same subtype is used for both the loop index variable and the array index. This is a common situation and it means the compiler can optimize out the run time checks normally required when accessing an array element. Since the value of Day can't possibly be outside the range of allowed array indexes, there is no need to check that Work_Hours(Day) is in bounds.

In many cases it is appropriate to give a name to an array type. This is necessary if you want to assign one array to another, compare two arrays for equality, or pass arrays into or out of subprograms. These operations require matching types, and with no way to talk about the type of array it isn't possible to declare two arrays with the same type. The following example illustrates:

```
procedure Example is
  type Buffer_Type is array(0..1023) of Character;
  B1 : Buffer_Type;
  B2 : Buffer_Type;
  B3 : array(0..1023) of Character;
begin
  B1 := B2; -- Fine. Entire array assigned.
  B1 := B3; -- Error. Types don't match. B3 has anonymous type.
end Example;
```

Because subtypes are defined dynamically the size and bounds of an array can also be defined dynamically. For example a declaration such as A : **array** (1.. N) **of** Natural is allowed even if the value of N is not known at compile time. For example, it could be a subprogram parameter. This feature means that many places where dynamic allocation is necessary in, for example, C++ can be handled in Ada without the use of explicit memory allocators.[3]

1.5.1 Unconstrained Array Types

One problem with the arrays I've described so far is that it is not possible to write a procedure that takes an array of unknown size. Arrays of different sizes have different types and strong typing will prevent them from being mixed. To get around this problem Ada has the concept of unconstrained array types. Such a type does not specify the bounds on the array as part of the type, allowing arrays with different bounds to be in the same type. However, the bounds must be specified when an actual array object is declared so the compiler knows how much memory to allocate for the object. The following example illustrates:

```
procedure Unconstrained_Array_Example is
  type Buffer_Type is array( Integer range <>) of Character;
  B1 : Buffer_Type;   -- Error. Must specify array bounds.
  B2 : Buffer_Type(  0..15);   -- Okay.
  B3 : Buffer_Type(  0..15);
  B4 : Buffer_Type (16..31);   -- Fine.
  B5 : Buffer_Type(  0..63);   -- No problem.

  procedure Process_Buffer(Buffer : in Buffer_Type) is
  begin
    for I in Buffer'Range loop
      -- Do something with Buffer(I)
```

[3] In fairness, C++ programmers routinely use library containers that hide the dynamic memory allocation they require.

```
    end loop;
  end Process_Buffer;

begin
  B2 := B3;  -- Fine. Types match and bounds compatible.
  B2 := B4;  -- Fine! Types match and lengths identical.
  B2 := B5;  -- Constraint_Error. Lengths don't match.

  Process_Buffer(B2);  -- Fine.
  Process_Buffer(B4);  -- Fine.
  Process_Buffer(B5);  -- Fine.
end Unconstrained_Array_Example;
```

Again there are several points to make about this example.

1. The symbol <> (pronounced "box") is intended to be a placeholder for information that will be filled in later. In this case it specifies that the index bounds on the array type Buffer_Type is unconstrained.

2. It is illegal to declare a variable with an unconstrained type without providing constraints. This is why the declaration of B1 is an error. However, the missing array bounds are specified in the other declarations. Notice that it is not necessary for array bounds to start at zero or one. Any particular value is fine as long as it is part of the index type mentioned in the array declaration.

3. It is fine to declare a subprogram, such as Process_Buffer, taking an unconstrained array type as an parameter. However, such a subprogram can't safely access the elements of the given array using specific index values like 1 or 2 because those values might not be legal for the array. Instead you need to use special array attributes. For example, Buffer'First is the first index value that is valid for array Buffer, and similarly for Buffer'Last.[4]. The attribute **Range** is shorthand for Buffer'First .. Buffer'Last and is quite useful in for loops as the example illustrates.

4. You might find it surprising that the assignment B2 := B4 is legal since the array bounds do not match. However, the two arrays have the same length so corresponding elements of the arrays are assigned. This is called *sliding semantics* because you can imagine sliding one array over until the bounds do match.

5. All the calls to Process_Buffer are fine. Inside the procedure Buffer'**Range** adapts itself to whatever array it is given and, provided the code is written with this in mind, the procedure works for arrays of any size.

1.5.2 Records

In C family languages composite objects containing components with different types are called structures. In Ada, and in many other languages, they are called records. All

[4]These attributes only return valid indexes if the array has a non-zero length

record types must have a name and thus must be declared before any objects of that type can be created. An example follows:

```
procedure Example is
  type Date is
    record
      Day   : Integer range 1 .. 31;
      Month : Integer range 1 .. 12;
      Year  : Natural;
    end record;

  Today    : Date := (Day => 1, Month => 7, Year => 2007);
  Tomorrow : Date;
begin
  Tomorrow := Today;
  Tomorrow.Day := Tomorrow.Day + 1;
end Example;
```

This procedure defines a type Date as a collection of three named components. Notice that the Day and Month components are defined as subtypes of Integer without the bother of naming the subtypes involved (anonymous subtypes are used). Today is declared to be a Date and initialized with a *record aggregate*. In this case named association is used to make it clear which component gets which initial value, however positional association is also legal.

Records can be assigned as entire units and their components accessed using the dot operator. Notice that if Tomorrow.Day + 1 creates a value that is outside the range 1 .. 31, a Constraint_Error exception will be raised, as usual when that out of bounds value is assigned to Tomorrow.Day. Obviously a more sophisticated procedure would check for this and adjust the month as necessary (as well as deal with the fact that not all months are 31 days long).

1.6 Packages

A package is a named container into which you can place subprograms, type definitions, and other entities that are related to each other. Packages can even contain other packages. In Ada packages are the primary way to organize and manage the various parts of a program.

Packages have two parts: a *specification* that declares the entities that are visible to the rest of the program, and a *body* that contains the implementation of those entities. The body can also contain private entities that can only be used inside the package. In most cases the specification is stored in a separate file from the body and can even be compiled separately. This makes the separation between a package's interface and implementation more rigorous in Ada than in many languages, encouraging the programmer to separate the process of software design from construction.

This formal separation makes it easier for different groups of people to work on the design and construction of a program since those groups would be working with different

sets of source files. At least that is the ideal situation. Alas, package specifications can have a private section, discussed later, that is technically part of the package's implementation. Thus, in some cases at least, interface and implementation end up in the same file after all.

As an example, consider a simple package for displaying text on a character mode terminal. The following example shows what the specification might look like. When using the GNAT compiler, package specifications are stored in files with a `.ads` extension using the same name as the name of the package. In this case, the file would be `screen.ads`.

```
package Screen is

   type Row_Type is range 1..25;
   type Column_Type is range 1..80;
   type Color_Type is (Black, Red, Green, Blue, White);

   procedure Set_Cursor_Position
      (Row : in Row_Type; Column : in Column_Type);
   procedure Get_Cursor_Position
      (Row : out Row_Type; Column : out Column_Type);


   procedure Set_Text_Color(Color : in Color_Type);
   procedure Get_Text_Color(Color : out Color_Type);

   procedure Print(Text : String );

end Screen;
```

There is normally no executable code in the specification; it only contains declarations.[5] The example above shows a few types being declared and then some procedures that make use of those types. Packages can also contain functions, of course.

Once the specification has been written code that makes use of the package can be compiled. Such a compilation unit contains a statement such as **with** Screen in its context clause. It can then, for example, refer to the type Screen.Row_Type and call procedure Screen.Print as you might expect. Note that a package specification can contain a context clause of its own if it needs, for example, types defined in some other package.

The package body, stored in a `.adb` file, contains the implementation of the various subprograms declared in the specification. The Ada compiler checks for consistency between the specification and the body. You must implement all subprograms declared in the specification and you must even use exactly the same names for the parameters to those subprograms. However, the body can define types and subprograms that are not declared in the specification. Such entities can only be used inside the package body; they are not visible to the users of the package. The following example shows an abbreviated version of `screen.adb`:

[5] The initializer of a variable, however, can contain executable code.

```
package body Screen is

   -- Not declared in the spec. This is for internal use only.
   procedure Helper is
   begin
     -- Implementation not shown.
   end Helper;

   -- All subprograms declared in spec must be implemented.
   procedure Set_Cursor_Position
      (Row : in Row_Type; Column : in Column_Type) is
   begin
     -- Implementation not shown.
   end Set_Cursor_Position ;

   -- etc...

end Screen;
```

The package body does not need to include a with clause for its own specification, however it can **with** other packages as necessary in order to gain access to the resources provided by those packages.

When building a large program the package specifications can all be written first, before any real coding starts. The specifications are thus the final output of the design phase of the software development life cycle. These specifications can be compiled separately to verify that they are free from syntax errors and that they are internally consistent. The compiler can verify that all the necessary dependencies are explicitly declared via appropriate **with** statements, and that all types are used in a manner that is consistent with their declarations.

Once the specifications are in place, the bodies can be written. Because a package body only depends on the specifications of the packages it uses, the bodies can all be written in parallel, by different people, without fear of inconsistency creeping into the program. Programmers could be forbidden to modify any specification files, for example by using appropriate access control features on a file server or version control system, requiring any such modifications to first be cleared by a designer. Surprisingly few programming languages enjoy this level of rigor in the management of code, but Ada was designed for large software projects and so its features are strong in this area.

1.6.1 Package Initialization

Packages sometimes contain internal variables that need to be initialized in some complex way before the package can be used. The package author could provide a procedure that does this initialization, but then there is a risk that it won't get called when it should. Instead, a package can define some initialization code of arbitrary complexity by introducing an executable section in the package body. The following example shows how this can be done.

```
with Other;
pragma Elaborate_All(Other);

package body Example is

   Counter : Integer;   -- Internal variable.

   procedure Some_Operation is
   begin
      -- Implementation not shown.
   end Some_Operation;

begin
  Counter := Other.Lookup_Initial ("database.vtc.vsc.edu");

end Example;
```

In this example, an internal variable Counter is initialized by calling a function in a supporting package Other. This initialization is done when the package body is elaborated. Notice that it is necessary in this example for the body of package Other to be elaborated first so that its initialization statements (if any) execute before the elaboration of the body of package Example. If that were not the case, then function Lookup_Initial might not work properly. An Ada program will raise the exception Program_Error at run time if packages get elaborated in an inappropriate order.

To address this potential problem a pragma is included in the context clause of package Example's body. Pragmas are commands to the compiler that control the way the program is built. The Ada standard defines a large number of pragmas and implementations are allowed to define others. In this case the use of pragma Elaborate_All forces the bodies of package Other, and any packages that package Other uses, to be elaborated before the body of package Example. Note that the **with** clauses will automatically cause the specifications of the withed packages to be elaborated first, but not necessarily the bodies. In this case it is essential to control the order in which the package bodies are elaborated. Hence the pragma is necessary.

Notice that Ada does not provide any facilities for automatically cleaning up a package when the program terminates. If a package has shutdown requirements a procedure must be defined for this purpose, and arrangements must be made to call that procedure at an appropriate time. In embedded systems, one of the major target application areas for Ada, programs often never end. Thus the asymmetric handling of package initialization and cleanup is reasonable in that context. Other programming languages do provide "module destructors" of some kind to deal with this matter more uniformly.

1.6.2 Child Packages

Being able to put code into various packages is useful, but in a large program the number of packages might also be large. To cope with this Ada, like many languages, allows its packages to be organized into a hierarchy of packages, child packages, grand

child packages, and so forth. This allows an entire library to be contained in a single package and yet still allows the various components of the library to be organized into different packages (or package hierarchies) as well. This is an essential facility for any language that targets the construction of large programs.

As an example, consider a hypothetical data compression library. At the top level we might declare a package Compress to contain the entire library. Package Compress itself might be empty with a specification that contains no declarations at all (and no body). Alternatively one might include a few library-wide type definitions or helper subprograms in the top level package.

One might then define some child packages of Compress. Suppose that Compress.Algo contains the compression algorithms and Compress. Utility contains utility subprograms that are used by the rest of the library but that are not directly related to data compression. Further child packages of Compress.Algo might be defined for each compression algorithm supported by the library. For example, Compress.Algo.LZW might provide types and subprograms related to the LZW compression algorithm and Compress.Algo.Huffman might provide types and subprograms related to the Huffman encoding compression algorithm. The following example shows a procedure that wishes to use some of the facilities of this hypothetical library.

```
with Compress.Algo.LZW;

procedure Hello is
  Compressor : Compress.Algo.LZW.LZW_Engine;
begin
  Compress.Algo.LZW.Process(Compressor, "Compress Me!");
end Hello;
```

In this example I assume the LZW package provides a type LZW_Engine that contains all the information needed to keep track of the compression algorithm as it works (presumably a record of some kind). I also assume that the package provides a procedure Process that updates the compressed data using the given string as input.

Clearly a package hierarchy with many levels of child packages will produce very long names for the entities contained in those deeply nested packages. This can be awkward, but Ada provides two ways to deal with that. You have already met one way: the **use** statement. By including use Compress.Algo.LZW in the context clause, the contents of that package are made directly visible and the long prefixes can be deleted. However, Ada also allows you to rename a package to something shorter and more convenient. The following example shows how it could look.

```
with Compress.Algo.LZW;

procedure Hello is
  package Squeeze renames Compress.Algo;
  Compressor : Squeeze.LZW.LZW_Engine;
begin
  Squeeze.LZW.Process(Compressor Compress Me!    );
end Hello;
```

The package Squeeze is not a real package, but rather just a shorter, more convenient name for an existing package. This allows you to reduce the length of long prefixes without eliminating them entirely. Although the Ada community encourages the use of long, descriptive names in Ada programs, names that are local to a single procedure can sensibly be fairly short since they have limited scope. Using the renaming facility of the language you can introduce abbreviated names for packages (or other kinds of entities) with limited scope without compromising the overall readability of the program.

Notice that the optimal names to use for entities in a package will depend on how the package itself is used. In the example above the package LZW provides a type LZW_Engine. In cases (such as shown) where fully qualified names are used the "LZW" in the name LZW_Engine is redundant and distracting. It might make sense to just name the type Engine yielding a fully qualified name of Compress.Algo.LZW.Engine. On the other hand, if use statements are used the name Engine by itself is rather ambiguous (what kind of engine is that?). In that case it makes more sense to keep the name LZW_Engine. Different programmers have different ideas about how much name qualification is appropriate and it leads to differences in the way names are selected.

The problem is that the programmer who writes a package is often different from the programmer who uses it and so incompatible naming styles can arise. Ada's renaming facility gives the using programmer a certain degree of control over the names that actually appear in his or her code, despite the names selected by the author of a library. The following example shows a more radical approach to renaming.

```
with Compress.Algo.LZW;

procedure Hello is
  subtype LZW_Type is Compress.Algo.LZW.LZW_Engine;
  procedure Comp(Engine : LZW_Type; Data : String)
       renames Compress.Algo.LZW.Process;
  Compressor : LZW_Type;
begin
  Comp(Compressor "Compress Me!");
end Hello;
```

Notice that while packages and subprograms (and also objects) can be renamed, types can not be renamed in this way. Instead you can introduce a subtype without any constraints as a way of effectively renaming a type.

In this simple example, the renaming declarations bulk up the code more than they save. However, in a longer and more complex situation they can be useful. On the other hand, you should use renaming declarations cautiously. People reading your code may be confused by the new names if it is not clear what they represent.

1.7 Abstract Data Types

Often it is desirable to hide the internal structure of a type from its users. This allows the designer of that type to change its internal structure later without affecting the code that uses that type. Ada supports this concept by way of private types. A type can be

declared as private in the visible part of a package specification. Users can then create variables of that type and use the built in assignment and equality test operations on that type. All other operations, however, must be provided in the package where the type is defined. Subprograms in the same package as the type have access to the type's internal representation. The following example shows a hypothetical package that provides a type representing calendar dates.

```
package Example is
   type Date is private;

   function Make(Year, Month, Day : Integer) return Date;
   function Get_Year(Today : Date) return Integer;
   -- etc...
   function Day_Difference(Future, Past : Date) return Integer;
   -- etc...

private
   type Date is
      record
         Y : Integer;
         M : Integer;
         D : Integer;
      end record;
end Example;
```

Note that in a more realistic version of this package, one might introduce different types for year, month, and day values in order to prevent programs from accidentally mixing up those concepts. However, in the interest of brevity I do not show that in this example. The type Date is declared private in the visible part of the specification. It is then later fully defined as a record in the private part.

Theoretically the full view of a type should not be in the specification at all; it is part of the package's implementation and thus should only be in the package body. However, Ada requires that private types be fully defined in package specifications as a concession to limited compiler technology. When the compiler translates code that declares variables of the private type, it needs to know how much memory to set aside for such variables. Thus while the programmer is forbidden to make use of the private type's internal structure, the compiler needs to do so. Note that some programming languages go further than Ada in this regard and move all such private information out of the interface definition. However, there is generally an execution performance penalty involved in doing this. In Ada it is possible to simulate such a feature using the so called "pimpl idiom." This method involves access types and controlled types and so is not described further here.

Note also that the full view of a private type does not need to be a record type. Although records are common in this situation, private types can be implemented as arrays or even just simple scalars like Integer or Float.

Private types do allow assignment and equality tests. However, in some cases it is desirable to disallow those things. Ada makes this possible using limited private types.

A declaration such as:

```
type Date is limited private;
```

tells the compiler to disallow built in assignment and equality tests for the specified type. This does not mean assignment is impossible; it only means that the package author must now provide a procedure to do assignment—if support for assignment is desired. Presumably that procedure would take whatever steps were necessary to make the assignment work properly. Calendar dates are not good candidates to make into a limited type. The component-wise assignment of one date to another is exactly the correct way to assign dates. Thus the built in record assignment operation is fine. Similar comments apply to the built in test for equality operation. However, types that represent resources outside the computer system (for example a network connection) can't reasonably be assigned. They should be declared limited private instead.

1.8 Strings

To be written...

1.9 Exceptions

Like many modern programming languages, Ada allows subprograms to report error conditions using exceptions. When an error condition is detected an exception is raised. The exception propagates to the dynamically nearest handler for that exception. Once the handler has executed the exception is said to have been handled and execution continues after the handler. Exceptions are not resumable. This model is largely the same as that used by C++ although the terminology is different. However, unlike C++, Ada exceptions are not ordinary objects with ordinary types. In fact, Ada exceptions don't have a type at all and exist outside the language's type system.

Exceptions are typically defined in a package specification. They are then raised in the package body under appropriate conditions and handled by the package users. The following example shows part of a specification for a Big_Number package that supports operations on arbitrary precision integers.

```
package Big_Number is
  type Number_Type is private;

  Divide_By_Zero : exception;

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

private
  -- Not shown.
end Big_Number;
```

Notice that Ada, like C++, allows operators to be overloaded. In Ada this is done by defining functions with names given by the operator names in quotation marks. This package also defines an exception that will be raised when one attempts to use Big_Number."/" with a zero divisor.

The implementation of the division operation might look, in part, as follows:

```
function "/"(Left, Right : Number_Type) return Number_Type is
begin
  -- Assume there is a suitable overloaded = operator.
  if Right = 0 then
    raise Divide_By_Zero;
  end if;

  -- Proceed knowing that the divisor is not zero.
end "/";
```

As you might guess, the **raise** statement aborts execution of the function immediately. A procedure that wishes to use Big_Number might try to handle this exception in some useful way. The following example shows the syntax:

```
procedure Example is
  use type Big_Number.Number_Type;
  X, Y, Z : Big_Number.Number_Type;
begin
  -- Fill Y and Z with interesting values.
  X := Y / Z;

exception
  when Big_Number.Divide_By_Zero =>
    Put_Line("Big Number division by zero!");

  when others =>
    Put_Line("Unexpected exception!");
end Example;
```

The block delimited by **begin** and **end** can contain a header **exception**. After that header a series of **when** clauses specify which exceptions the block is prepared to handle. The special clause **when others** is optional and is used for all exceptions that are otherwise not mentioned.

If the normal statements in the block execute without exception, control continues after the block (skipping over the exception clauses). In the previous example the procedure returns at that point. If an exception occurs in the block, that exception is matched against those mentioned in the when clauses. If a match is found the corresponding statements are executed and then control continues after the block. If no match is found, the block is aborted and the next dynamically enclosing block is searched for a handler instead. If you are familiar with C++ or Java exceptions none of this should be surprising.

The previous example also shows an interesting detail related to operator overloading. The example assumes that there is a context clause of **with** Big_Number (not shown)

but no **use** statement. Thus the division operator is properly named Big_Number."/". Unfortunately it can't be called using the infix operator notation with that name. There are several ways to get around this. One could include a **use** Big_Number in the context clause or in the procedure's declarative part. However that would also make all the other names in package Big_Number directly visible and that might not be desired. An alternative is to introduce a local function named "/" that is just a renaming (essentially an alias) of the function in the other package. As we have seen Ada allows such renaming declarations in many situations, but in this case, every operator function would need a corresponding renaming and that could be tedious.

Instead the example shows a more elegant approach. The **use type** declaration tells the compiler that the primitive operations of the named type should be made directly visible. I will discuss primitive operations in more detail in the section on object oriented programming. In this case, the operator overloads that are declared in package Big_Number are primitive operations. This method allows all the operator overloads to be made directly visible with one easy statement, and yet does not make every name in the package directly visible.

The Ada language specifies four predefined exceptions. These exceptions are declared in package Standard (recall that package Standard is effectively built into the compiler) and thus directly visible at all times. The four predefined exceptions with their use are described as follows:

- Constraint_Error. Raised whenever a constraint is violated. This includes going outside the bounds of a subtype (or equivalently the allowed range of an array index) as well as various other situations.
- Program_Error. Raised when certain ill formed programs that can't be detected by the compiler are executed. For example, if a function ends without executing a return statement, Program_Error is raised.
- Storage_Error. Raised when the program is out of memory. This can occur during dynamic memory allocation, or be due to a lack of stack space when invoking a subprogram. This can also occur during the elaboration of a declarative part (for example if the dynamic bounds on an array are too large).
- Tasking_Error. Raised in connection with certain tasking problems.

Most of the time the predefined exceptions are raised automatically by the program under the appropriate conditions. It is legal for you to explicitly raise them if you choose, but it is recommended that you define your own exceptions for your code. Note that the Ada standard library defines some additional exceptions as well. Those exceptions are not really predefined because, unlike the four above, they are not built into the language itself.

1.10 Discriminated Types

To be written...

1.11 Generics

Statically checked strongly typed languages like Ada force subprogram argument types and parameter types to match at compile time. This is awkward in situations where the same sequence of instructions could be applied to several different types. To satisfy the compile time type checking requirements, it is necessary to duplicate those instructions for each type being used. To get around this problem, Ada allows you to define generic subprograms and generic packages where the types used are parameters to the generic unit. Whenever a new version of the generic unit is needed, it is *instantiated* by the compiler for specific values of the type parameters.

Ada generics are very similar in concept and purpose to C++ templates. Like C++ templates, generics are handled entirely at compile time; each instantiation generates code that is specialized for the types used to instantiate it. This is different than generics in Java which are resolved at run time and that use a common code base for all instances. However, Ada generics and C++ templates also differ in some important ways. First Ada requires the programmer to explicitly instantiate a generic unit using special syntax. In C++ instantiations are done implicitly. In addition, Ada generics do not support explicit specialization or partial specialization, two features of C++ templates that are used in many advanced template libraries.

There are pros and cons to Ada's approach as compared to C++'s method. With Ada, it is very clear where the instantiation occurs (since it is shown explicitly in the source code). However, C++'s method enables advanced programming techniques (template meta-programming) that Ada can't easily replicate.

The other major difference between Ada generics and C++ templates is that Ada allows the programmer to specify the necessary properties of the types used to instantiate a generic unit. In contrast in C++ one only says that the parameter is a type. If a type is used to instantiate a template that doesn't make sense, the compiler only realizes that when it is actually doing the instantiation. The resulting error messages can be very cryptic. In contrast the Ada compiler can check before it begins the instantiation if the given types are acceptable. If they are not, it can provide a much clearer and more specific error message.

To illustrate how Ada generics look, consider the following example that shows the specification of a generic package containing a number of sorting procedures.

```
generic
  type Element_Type is private;
  with function "<"(Left, Right : Element_Type) return Boolean is <>;
package Sorters is
  type Element_Array is array(Natural range <>) of Element_Type;

  procedure Quick_Sort(Sequence : in out Element_Array);
  procedure Heap_Sort(Sequence : in out Element_Array);
  procedure Bubble_Sort(Sequence : in out Element_Array);
end Sorters;
```

Notice that the specification has a generic header that defines the parameters to this

generic unit. The first such parameter is a type that will be called Element_Type in the scope of the generic package. It is declared as a private type to indicate that the package will only (by default) be able to assign and compare for equality objects of that type. It is thus possible to instantiate this package for private types, but non-private types like Integer, arrays, and records also have the necessary abilities and can be used. However, a limited type (for example, a type declared as **limited private** in some other package) can not be used.

This generic package also requires its user to provide a function that can compare two Element_Type objects. That function is called "<" in the scope of the package, but it could be called anything by the user. It must, however, have the same profile (the same number and type of parameters). The <> symbol at the end of the function parameter indicates that the compiler should automatically fill in the required function if one is available at the instantiation point. This allows the user to instantiate the package using just one type argument if the default "<" for that type is acceptable.

Inside the implementation of the generic package, Element_Type can be used as a private type except that it is also permitted to use "<" to compare two Element_Type objects. No other operations on Element_Type objects are allowed to guarantee the package will instantiate correctly with any type the user is allowed to use.

The following example shows how this package might be used:

```
with Sorters;

procedure Example is
  package Integer_Sorters is
    new Sorters(Element_Type => Integer, "<" => Standard."<");

  Data : Integer_Sorters.Element_Array;
begin
  -- Fill Data with interesting information.

  Integer_Sorters.Quick_Sort(Data);
end Example;
```

Notice the first line in the declarative region that instantiates the generic unit. In the example, named parameter association is used to bind the generic unit's arguments to its parameters. The strange looking construction "<"=> Standard."<" is an association between the generic parameter "<" (the comparison function) and the operator < that applies to integers (declared in package Standard). It would be more typical for the right side of this association (and even the left side as well) to be named functions.

In fact the second parameter of the instantiation is not actually necessary in this case because, due to the <> symbol used in the generic parameter declaration, the compiler will fill in the proper function automatically. However, it is still possible to provide the function explicitly in cases where some different function is desired.

The name Integer_Sorters is given to the specific instance created. That name can then be used like the name of any other package. In fact, it is legal (and common) to follow a generic instantiation with a **use** statement to make the contents of the newly

instantiated package directly visible.

Ada also allows procedures and functions to be generic using an analogous syntax.

This example is fairly simple in that the generic type parameter has very limited abilities. It is also possible to specify that the parameter is a discrete type (thus allowing the use of 'First, 'Last, 'Succ and 'Pred attributes), an access type, a tagged type, a modular type, a floating point type, and various other possibilities. In addition, Ada's generic facility allows generic units to be parameterized on values (so called "non-type" parameters in C++) and variables. I refer you to one of the references for more information on generic units in Ada. The discussion here is only scratching the surface of this large and important topic.

1.12 Access Types

Ada, like many languages, allows you to create objects that refer to other objects. It also allows you to create objects that have a lifetime extending beyond that of the scope in which they are created. Both of these capabilities are important and many programming techniques depend on them. C and C++ allow the programmer to use arbitrary pointers, implemented as simple memory addresses, in any way desired. While this is very powerful, it is also very dangerous. C and C++ programs suffer from many bugs and security vulnerabilities related to the unrestricted use of pointers in those languages.

In Ada pointer types are called *access types*. Like C and C++ pointer variables, *access variables* can be manipulated separately from the objects to which they point. However, unlike C and C++ pointer types, access types in Ada have a number of restrictions on their use that are designed to make them safer.

Actually, the history of access types in Ada is quite interesting. In Ada 83 access types were very limited but also very safe. However, experience with Ada showed that the limitations were too great. Ada 95 removed some of the limitations while still managing to keep the safety (at the expense of complicating the language). Yet even after these changes, access types were still not as flexible as desired. Ada 2005 removed yet more limitations but required, in certain cases, run time checking to be done to ensure safety. In this tutorial I will not describe these issues in detail. My focus here is on the basic use of access types.

Access types can be named or anonymous. I will only consider named access types; anonymous access types have special properties that are outside the scope of this tutorial.

For example the declaration:

```
type Integer_Access is access Integer;
```

declares Integer_Access as a type suitable for accessing, or pointing at, integers. Notice that in this tutorial I have used a suffix of "Type" when naming a type. In this case, however, I use a suffix of "Access" to emphasize the nature of the access type. This is, of course, just a convention.

Once the access type has been declared, variables of that type can then be declared in the usual way.

```
P : Integer_Access ;
```

Ada automatically initializes access variables to the special value **null** if no other initialization is given. Thus access variables are either null or they point at some real object. Indeed, the rules of Ada are such that, under appropriate circumstances, dangling pointers are not possible. Access variables can be copied and compared like any other variable. For example if P1 and P2 are access variables than P1 = P2 is true if they both point at the same object. To refer to the object pointed at by an access variable you must use the special .**all** operation.

```
P.all := 1;
```

The .**all** syntax plays the same role in Ada as the indirection operator (the *) plays in C and C++. The use of .**all** may seem a little odd in this context, but understand that most of the time access types are pointers to aggregate entities such as arrays or records. In that case, the components of the array or record pointed at can be accessed directly by using the component selection operation on the access variable itself. This is shown as follows:

```
type Date is
  record
    Day, Month, Year : Integer ;
  end record;
type Date_Access is access Date;

D1, D2 : Date_Access;

...

D1.Day := 1;       -- Accesses the Day member of referenced Date.
D1     := D2;      -- Causes D1 to point at same object as D2.
D1.all := D2.all;  - Copies Date objects.
```

In this case the .**all** syntax is very natural. It shows that you are accessing all the members of the referenced object at the same time. It is important to notice that Ada normally "forwards" operations applied to the access variable to the referenced object. Thus D1.Day in the example means the Day component of the object pointed at by D1. Only operations that are meaningful for the access type itself are not forwarded. Thus D1 := D2 copies the access values. To copy the objects pointed at by the access variables one must use the .**all** syntax. I should also note that syntax such as D1.**all**.Day, while verbose, is also legal. The .**all** dereferences D1. The result is a date record so the selection of the Day component is meaningful.

Access variables that refer to arrays also allow the normal array operations to be forwarded to the referenced object as the following example illustrates:

```
type Buffer_Type is array(0..1023) of Character;
type Buffer_Access is access Buffer_Type;
B1, B2 : Buffer_Access ;
```

```
...

B1      := B2;      -- Copies only the access values.
B1.all  := B2.all   -- Copies arrays.
B1(0)   := 'X'      -- Forwards index operation.
for I in B1'Range loop   -- Forwards 'Range attribute.
   ...
end loop;
```

Because of forwarding, using access types is generally quite convenient in Ada. However, you must keep in mind that operations that are meaningful for the access types themselves will be applied directly to the access variable and are not forwarded.

So far we've seen how to declare and use access types. How does one get an access variable to point at another object in the first place? In Ada 83 there was only one way: using the new operation. For example:

```
P := new Integer'(0);
```

dynamically allocates an integer, initializes that integer to zero, and then assigns the resulting access value to P. Here I assume P has an access to integer type. Notice that the argument to new has the form of a qualified expression (the apostrophe is required). Also as with dynamically allocated objects in other languages, the lifetime of the object created in this way extends beyond the lifetime of the access variable used to point at it.

1.12.1 Garbage Collection?

In many languages dynamically allocated objects are automatically reclaimed when they can no longer be accessed. For example, in Ada parlance, when all access variables pointing at an object go out of scope the object that was pointed at by those variables can no longer be referenced and the memory it uses should be made available again. The process of reclaiming such memory is called *garbage collection*.

In Ada garbage collection is optional. Implementations are allowed to provide it, but they are not required to do so. This may seem surprisingly wishy-washy for a language that endeavors to support reliable and portable programming. The problem is that Ada also endeavors to support low level embedded systems and real time programming. In such environments garbage collection is widely considered problematic. It is difficult to meet real time objectives if a complex garbage collection algorithm might run at any time. Also the space overhead of having a garbage collector in the run time system might be unacceptable for space constrained embedded devices. Advances in garbage collection technology and machine capabilities have made some of these concerns less pressing today than they were when Ada was first designed. However, these matters are still important. Thus Ada allows, but does not require garbage collection.

This presents an immediate problem for programmers interested in portability. If the implementation does not collect its garbage and the programmer takes no steps to manually reclaim allocated objects, the program will leak memory. This is a disaster

for long running programs like servers. Thus for maximum portability one must assume that garbage collection is not done and take steps accordingly. In fact, most Ada implementations do not provide garbage collection, so this is a very realistic concern.

The Ada library provides a generic procedure named Unchecked_Deallocation that can be used to manually deallocate a dynamically allocated object. Unfortunately the use of Unchecked_Deallocation can violate important safety properties the language otherwise provides. In particular, if you deallocate an object while an access variable still points at it, any further use of that access variable will result in erroneous behavior. As a service Unchecked_Deallocation will set the access variable you give it to **null** causing future use of that access variable to result in a well defined exception. However, there might be other access variables that point at the same object and Unchecked_Deallocation cannot, in general, know about all of them.

If your program never uses Unchecked_Deallocation then all access variables are either **null** or point at a real object; dangling pointers are impossible. However, your program might also leak memory if the Ada implementation does not provide garbage collection. Thus in most real programs Unchecked_Deallocation is used.

This is an example of where Ada compromises safety for the sake of practical reality. In fact, there are several other "Unchecked" operations in Ada that are used to address certain practical concerns and yet introduce the possibility of unsafe programs. Since every such operation starts with the word "Unchecked" it is an simple matter to search an Ada program for occurrences of them. This feature makes reviewing the unchecked operations easier.

Show an example of Unchecked_Deallocation...

1.13 Command Line Arguments

To be written...

1.14 Object Oriented Programming

To be written...

1.15 Tasking

Many programs can be naturally described as multiple, interacting, concurrent threads of execution. Programming environments that provide direct support for concurrency are thus very useful. Such support can be offered by the operating system or by the programming language or some combination of the two.

The classic way in which operating systems support concurrency is by allowing multiple independent processes to run simultaneously. This method has the advantage of offering good isolation between the processes so that if one crashes the others are not necessarily affected. On the other hand, communication between processes tends to have high overhead. Modern operating systems also usually allow programs to be written that

have multiple threads of control executing in the same process. This *thread level concurrency* is harder to program correctly but has reduced overhead as compared to the older style *process level concurrency*.

In some environments offering thread level concurrency the programmer must invoke subprograms in a special library to create and synchronize threads. Such an approach requires relatively little support from the programming language but it tends to be error prone. Another approach is to build support for thread level concurrency into the programming language itself. This allows the compiler to take care of the low level details of thread management, freeing the programmer to focus on other aspects of the program's design. Ada uses this second approach and supports concurrent programming as a language feature.

The unit of execution in Ada is called a *task*. The main program executes in a task called the *environment task*. You can create additional tasks as appropriate to meet your application's needs. Like a package each task has a specification and a body. In the simplest case a task specification only needs to declare that the task exists. The following example illustrates the basic syntax.

```
with Ada.Text_IO;
with Helper;

procedure Main is

      Specification of nested task.
  task Nag;

     Body of nested task.
  task body Nag is
  begin
    for I in 1 .. 100 loop
      Ada.Text_IO.Put_Line("Hello");
      delay 10.0;
    end loop;
  end Nag;

begin
  Helper.Something_Useful;
end Main;
```

In this example a task Nag is both specified and defined in the declarative part of the main program. The task simply prints "Hello" one hundred times with a 10 second delay between each line of output. While the task does this important work, the main program simultaneously executes the useful function of the program.

It is important to understand that the task starts executing automatically as soon as the enclosing subprogram begins. It is not necessary to explicitly start the task. Furthermore the enclosing subprogram will not return until the task has completed. Care must be taken to ensure that the task eventually ends. If the task never ends the enclosing subprogram will never return.

Because the task is nested inside another program unit it has access to the local variables and other entities declared in the enclosing unit above the task's definition. This gives a way for the task to share information with the enclosing unit but beware that sharing data in this manner is difficult and error prone. As I will describe shortly Ada provides much more robust ways for tasks to communicate.

The example above shows a task nested inside a procedure. Tasks can also be nested inside functions or even inside each other. This allows you to create a task to assist in the execution of any subprogram without anyone outside your subprogram being aware that tasks are involved. You can also create tasks inside a package body that support the operation of that package. Be aware, however, that if you do create a task inside a library package body you need to arrange for that task to eventually end or else the program will never terminate.

Finish me...

1.16 Container Library

To be written...

1.17 Low Level Programming

To be written...

2 Huffman Encoding

To better understand the features of Ada, it is helpful to work through a small but realistic program using the language. In this section I will design and implement a simple file compression utility that uses the Huffman encoding compression algorithm. Although Huffman encoding is not particularly effective by itself, it is good enough to provide a usable utility for some applications. It also provides a good balance of complexity and simplicity for this exercise.

2.1 The Algorithm

Huffman encoding takes advantage of the fact that in most files some byte values are much more common than others. By assigning short binary codes to the most commonly occurring values the file can be made smaller. One consequence, however, of assigning short codes to some values is that other values must necessarily be assigned long binary codes. This is necessary in order to properly distinguish between all possible values.

For example, assume that the most commonly occurring value is assigned a single bit code of 1. All other values must necessarily have codes starting with 0 so that they can be properly recognized. There could be as many as 255 such values, requiring 8 additional bits (for a total of nine bits) to distinguish those values among themselves.

Huffman encoding finds a way to assign variable length codes to the byte values to minimize the overall file size. Specifically it uses byte value frequency information to make the code assignments. That is, commonly occurring bytes are assigned relatively short codes and infrequently occurring bytes are assigned correspondingly longer codes. Itt can be shown that Huffman encoding is optimal; no other compression method that does not take advantage of correlations can provide more compression. Huffman encoding is also an example of a greedy algorithm and is often discussed in algorithms textbooks as such.

The algorithm has three phases:

1. Scan the input and count the number of times each byte value occurs.
2. Construct a *code tree* that reflects the relative frequency of the byte values and assign the variable length codes to each value.
3. Rewrite the input substituting the codes assigned in step #2 for the byte values.

One significant disadvantage to Huffman encoding is that the input must be scanned completely to do the frequency analysis before any output can be written. In some applications the entire input is not known initially (for example: streaming data or user

input) and that is a major problem for this method. In such cases the compressor can sometimes make an educated guess about the relative frequencies of the byte values, based perhaps on past observations of similar input, but the results will then be only approximate. This is not an issue in this case; I am interested in compressing files and I have access to the entire file before I have to compress any of it.

The output of the first phase is, conceptually, a table with one entry for each byte value (256 entries in all). Associated with each entry is a count of the number of times that value appeared in the input. For example, assume the algorithm is applied to a file containing only the letters 'A' through 'E'. The table output by the first phase might look as follows:

Show table...

The table elements become the leaves of a tree. Pseudo code for the tree building algorithm is shown below. Each pass of the while loop combines the counts from the active nodes with the smallest and next smallest count. Those nodes are then removed from the active list and the new node is added to the active list (where its combined count is then considered in later passes). Thus each pass of the while loop reduces the size of the active list by one. Eventually only a single active node is left and that node is the root of the code tree.

```
WHILE <There is more than one active node> LOOP
  <Find nodes x, y with the smallest and next smallest counts>
  <Create a new node z that combines the count values;
     Point z at x and y>
  <Remove x and y from the set of active nodes>
  <Add z to the set of active nodes>
END
```

For example using the initial counts as shown above, the result after the first past of the while loop is as follows:

Show figure...

In the next pass of the while loop the nodes with counts 42 (smallest) and 45 (next smallest) will be combined. After the loop finishes executing in the sample above the code tree looks like:

Show figure...

Notice that the nodes with large counts, such as the node for 'A' are combined late in the process. Such nodes thus end up being a short distance from the root of the code tree. This is the basis for their short codes. Notice also that the count in the root node ends up being the same as the total number of bytes in the input. This can be used as a validity check on the code tree construction process.

After the code tree has been constructed, codes are assigned to the original byte values by assigning a 1 or 0 bit to each link in the code tree. The assignment can be arbitrary without affecting the correctness of the result, but for the sake of choosing something, I will assign a 1 bit to the child node with the larger count value. After the bit assignments, the code tree becomes:

Show figure...

The code for a byte value is then read from the root, assigning bits from left to right in the code as each link in the code tree is traversed. In my example this yields the codes as shown below:

Show table of assigned codes...

Notice how the codes for the most commonly occurring byte values ('A' and 'C') are short while the codes for the most infrequently occurring byte values ('B' and 'D') are long. Notice also that there is no ambiguity in the encoding. For example, a bit sequence such as 100110111000 can only be interpreted as the byte sequence "ABCAAE." Notice also that if ordinary 8 bit bytes are used the string "ABCAAE" would consume 6 bytes of space. However, the bit sequence 100110111000 is only 12 bits (1.5 bytes) long.

2.2 Implementation Notes

Before looking at the detailed design of the Huffman compression program I want to point out a few issues of interest. Keep these issues in mind when you review the program's design and the source code of the program itself.

1. Both the input and output files have to be processed as raw binary bytes without structure. To make the program general, it should not concern itself with the meaning of the input file's contents.

2. The construction of the code tree can't start until the input file has been fully analyzed. Similarly no output can be written until the code tree is fully constructed and code sequences have been assigned. This means there is little opportunity for parallelism among these phases of execution.

3. The pointers in a code tree node sometimes point at other code tree nodes (presumably allocated dynamically) and sometimes point at the original table of counter values.

4. An auxiliary data structure of some kind will be necessary to keep track of which code tree nodes are currently "active."

5. Since the code sequences have variable length, writing the output file is tricky. A way must be provided to write partial bytes and to write long bit sequences that span multiple bytes.

6. For the compressed file to be decompressed properly the receiver will need to build exactly the same code tree as the sender. One way to allow the receiver to do this would be for the sender to transmit the original counts in a header on the compressed file. If 32 bit counts are used, this adds 1024 bytes of overhead to the compressed file.

7. In a realistic example some code sequences can be very long (one or two hundred bits in an extreme case). It may be appropriate to use a variable length data structure to handle them.