Ada-95: A guide for C and C++ programmers

by Simon Johnston

Ada 95

The Language For A Complex World

1995

Welcome

... to the Ada guide especially written for C and C++ programmers.

Summary

I have endeavered to present below a tutorial for C and C++ programmers to show them what Ada can provide and how to set about turning the knowledge and experience they have gained in C/C++ into good Ada programming. This really does expect the reader to be familiar with C/C++, although C only programmers should be able to read it OK if they skip section 3.

My thanks to S. Tucker Taft for the mail that started me on this.

Contents

Introduction.

This document is written primarily for C and C++ programmers and is set out to describe the Ada programming language in a way more accessible to them. I have used the standard Ada documentation conventions, code will look like `this` and keywords will look like **`this`**. I will include references to the Ada Reference Manual in braces and in italics, *{1.1}*, which denotes section 1.1. The ARM is reference 1 at the end of this document. Another useful reference is the Lovelace on-line tutorial which is a great way to pick up Ada basics.

I will start out by describing the Ada predefined types, and the complex types, and move onto the simple language constructs. Section 2 will start to introduce some very Ada specific topics and section 3 describes the new Ada-95 Object Oriented programming constructs. Section 5 describes the Ada tools for managing concurrency, the task and protected types, these are worth investing some time getting to grips with. Section 6 is a tour of the Ada IO library and covers some of the differences in concept and implementation between it and C stdio.

Please feel free to comment on errors, things you don't like and things you would like to see. If I don't get the comments then I can't take it forward, and the question you would like answered is almost certainly causing other people problems too.

If you are new to Ada and do not have an Ada compiler handy then why not try the GNAT Ada compiler. This compiler is based on the well known GCC C/C++ and Objective-C compiler and provides a high quality Ada-83 and Ada-95 compiler for many platforms. Here is the FTP site (`ftp://cs.nyu.edu/pub/gnat`) see if there is one for you.

Document Status.

This document is still under revision and I receive a number of mails asking for improvements and fixing bugs and spelling mistakes I have introduced. I will try and keep this section up to date on what needs to be done and what I would like to do.

Current Status

Section 2 More on 2.3 (data hiding) and 2.4 (Hierarchical packages)

Section 3 First issue of this section, 3.6, 3.7, 3.8 and 3.9 have additional work planned. They may also require re-work pending comments.

Section 5 Section 5.3 (streams) not yet done.

Section 6 New sections to be added for each language.

Section 7 Major re-work following comments from Bill Wagner, 7.2.7 added, requires some more words, and section 7.3 requires more justification etc.

Wish List

I would like to use a consistant example throughout, building it up as we go along. The trouble is I don't think I have space in an HTML page to do this.

1

Ada Basics.

This section hopes to give you a brief introduction to Ada basics, such as types, statements and packages. Once you have these you should be able to read quite a lot of Ada source without difficulty. You are expected to know these things as we move on so it is worth reading.

One thing before we continue, most of the operators are similar, but you should notice these differences:

Operator	C/C++	Ada
Assignment	=	:=
Equality	==	=
NonEquality	!=	/=
PlusEquals	+=	
SubtractEquals	-=	
MultiplyEquals	*=	
DivisionEquals	/=	
OrEquals	|=	
AndEquals	&=	
Modulus	%	mod
Remainder		rem
AbsoluteValue		abs
Exponentiation		**
Range		. .

One of the biggest things to stop C/C++ programmers in their tracks is that Ada is case insensitive, so **begin BEGIN Begin** are all the same. This can be a problem when porting case sensitive C code into Ada.

Another thing to watch for in Ada source is the use of ’ the tick. The tick is used to access attributes for an object, for instance the following code is used to assign to value a the size in bits of an integer.

```
int a = sizeof(int) * 8;

a : Integer := Integer’Size;
```

Another use for it is to access the attributes `First` and `Last`, so for an integer the range of possible values is `Integer’First` to `Integer’Last`. This can also be applied to arrays so if you are passed an array and don't know the size of it you can use these attribute values to range over it in a loop (see section 1.1.5 on page 11). The tick is also used for other Ada constructs as well as attributes, for example character literals, code statements and qualified expressions (1.1.8 on page 16).

1.1 C/C++ types to Ada types.

This section attempts to outline how to move C/C++ type declarations into an Ada program and help you understand Ada code. Section 1.1.8 introduces some Ada specific advanced topics and tricks you can use in such areas as bit fields, type representation and type size.

Note that 'objects' are defined in reverse order to C/C++, the object name is first, then the object type, as in C/C++ you can declare lists of objects by seperating them with commas.

```
int i;
int a, b, c;
int j = 0;
int k, l = 1;
i : Integer;
a, b, c : Integer;
j : Integer := 0;
k, l : Integer := 1;
```

The first three declarations are the same, they create the same objects, and the third one assigns j the value 0 in both cases. However the fourth example in C leaves k undefined and creates l with the value 1. In the Ada example it should be clear that both k and l are assigned the value 1.

Another difference is in defining constants.

```
const int days_per_week = 7;
days_per_week : constant Integer := 7;
days_per_week : constant := 7;
```

In the Ada example it is possible to define a constant without type, the compiler then chooses the most appropriate type to represent it.

1.1.1 Declaring new types and subtypes.

Before we delve into descriptions of the predefined Ada types it is important to show you how Ada defines a type.

Ada is a strongly typed language, in fact possibly the strongest. This means that its type model is strict and absolutely stated. In C the use of typedef introduces a new name which can be used as a new type, though the weak typing of C and even C++ (in comparison) means that we have only really introduced a very poor synonym. Consider:

```
typedef int INT;
INT a;
int b;
a = b; // works, no problem
```

The compiler knows that they are both ints. Now consider:

```
type INT is new Integer;
a : INT;
b : Integer;
a := b; -- fails.
```

The important keyword is `new`, which really sums up the way Ada is treating that line, it can be read as "a new type `INT` has been created from the type `Integer`", whereas the C line may be interpreted as "a new name `INT` has been introduced as a synonym for `int`".

This strong typing can be a problem, and so Ada also provides you with a feature for reducing the distance between the new type and its parent, consider:

```
subtype INT is Integer;
a : INT;
b : Integer;
a := b; -- works.
```

The most important feature of the subtype is to constrain the parent type in some way, for example to place an upper or lower boundary for an integer value (see section below on ranges).

1.1.2 Simple types, Integers and Characters.

We have seen above the Integer type, there are a few more with Ada, these are listed below.

Integer, Long_Integer etc. Any Ada compiler must provide the Integer type, this is a signed integer, and of implementation defined size. The compiler is also at liberty to provide `Long_Integer, Short_Integer, Long_Long_Integer` etc as needed.

Unsigned Integers Ada does not have a defined unsigned integer, so this can be synthesised by a range type (see section 1.1.5), and Ada-95 has a defined package, `System.Unsigned_Types` which provide such a set of types.
Ada-95 has added a **modular** type which specifies the maximum value, and also the feature that arithmatic is cyclic, underflow/overflow cannot occur. This means that if you have a modular type capable of holding values from 0 to 255, and its current value is 255, then incrementing it wraps it around to zero. Contrast this with range types (previously used to define unsigned integer types) in section 1.1.5 below. Such a type is defined in the form:

```
type BYTE is mod 256;
type BYTE is mod 2**8;
```

The first simply specifies the maximum value, the second specifies it in a more 'precise' way, and the 2**x form is often used in system programming to specify bit mask types. Note: it is not required to use 2**x, you can use any value, so 10**10 is legal also.

Character *{3.5.2}* This is very similar to the C char type, and holds the ASCII character set. However it is actually defined in the package `Standard` {A.1} as an enumerated type (see section 1.1.5). There is an Ada equivalent of the C set of functions in `ctype.h` which is the package `Ada.Characters.Handling`.
Ada Also defines a `Wide_Character` type for handling non ASCII character sets.

Boolean ***{3.5.3}*** This is also defined in the package `Standard` as an enumerated type (see below) as (`FALSE, TRUE`).

1.1.3 Strings. *{3.6.3}*

Heres a god send to C/C++ programmers, Ada has a predefined String type (defined again in `Standard`). There is a good set of Ada packages for string handling, much better defined than the set provided by C, and Ada has a & operator for string concatenation.

As in C the basis for the string is an array of characters, so you can use array slicing (see below) to extract substrings, and define strings of set length. What, unfortunatly, you cannot do is use strings as unbounded objects, hence the following.

```
type A_Record is
 record
  illegal : String;
  legal   : String(1 .. 20);
 end record;
procedure check(legal : in String);
```

The illegal structure element is because Ada cannot use ’unconstrained’ types in static declarations, so the string must be constrained by a size. Also note that the lower bound of the size must be greater than or equal to 1, the C/C++ `array[4]` which defines a range `0..3` cannot be used in Ada, `1..4` must be used.

One way to specify the size is by initialisation, for example:

```
Name : String := "Simon";
```

is the same as defining `Name` as a `String(1..5)` and assigning it the value `"Simon"` seperatly..

For parameter types unconstrained types are allowed, similar to passing `int array[]` in C.

To overcome the constraint problem for strings Ada has a predefined package `Ada.Strings.Unbounded` which implements a variable length string type.

1.1.4 Floating *{3.5.7}* and Fixed *{3.5.9}* point.

Ada has two non-integer numeric types, the floating point and fixed point types. The predefined floating point type is `Float` and compilers may add `Long_Float`, etc. A new Float type may be defined in one of two ways:

```
type FloatingPoint1 is new Float;
type FloatingPoint2 is digits 5;
```

The first simply makes a new floating point type, from the standard `Float`, with the precision and size of that type, regardless of what it is.

The second line asks the compiler to create a new type, which is a floating point type "of some kind" with a minimum of 5 digits of precision. This is invaluable when doing numeric intensive operations and intend to port the program, you define exactly the type you need, not what you think might do today.

If we go back to the subject of the tick, you can get the number of digits which are actually used by the type by the attribute `'Digits`. So having said we want a type with minimum of 5 digits we can verify this:

```
number_of_digits : Integer := FloatingPoint2'Digits;
```

Fixed point types are unusual, there is no predefined type 'Fixed' and such type must be declared in the long form:

```
type Fixed is delta 0.1 range -1.0..1.0;
```

This defines a type which ranges from -1.0 to 1.0 with an accuracy of 0.1. Each element, accuracy, low-bound and high-bound must be defined as a real number.

There is a specific form of fixed point types (added by Ada-95) called decimal types. These add a clause digits, and the range clause becomes optional.

```
type Decimal is delta 0.01 digits 10;
```

This specifies a fixed point type of 10 digits with two decimal places. The number of digits includes the decimal part and so the maximum range of values becomes -99,999,999.99…+99,999,999.99

1.1.5 Enumerations *{3.5.1}* and Ranges.

Firstly enumerations. These are not at all like C/C++s enums, they are true sets and the fact that the Boolean type is in fact:

```
type Boolean is (FALSE, TRUE);
```

should give you a feeling for the power of the type.

You have already seen a range in use (for strings), it is expressed as `low .. high` and can be one of the most useful ways of expressing interfaces and parameter values, for example:

```
type Hours   is new Integer range 1 .. 12;
type Hours24 is range 0 .. 23;
type Minutes is range 1 .. 60;
```

There is now no way that a user can pass us an hour outside the range we have specified, even to the extent that if we define a parameter of type `Hours24` we cannot assign a value of `Hours` even though it can only be in the range. Another feature is demonstrated, for `Hours` we have said we want to restrict an `Integer` type to the given range, for the next two we have asked the compiler to choose a type it feels appropriate to hold the given range, this is a nice way to save a little finger tapping, but should be avoided Ada provides you a perfect environment to specify precisely what you want, use it the first definition leaves *nothing* to the imagination.

Now we come to the rules on subtypes for ranges, and we will define the two `Hours` again as follows:

```
type Hours24  is new range 0..23;
subtype Hours is Hours24 range 1..12;
```

This limits the range even further, and as you might expect a subtype cannot extend the range beyond its parent, so `range 0 .. 25` would have been illegal.

Now we come to the combining of enumerations and ranges, so that we might have:

```
type All_Days is (Monday, Tuesday, Wednesday, Thursday,
                  Friday, Saturday, Sunday);
subtype Week_Days is All_Days range Monday .. Friday;
subtype Weekend is All_Days range Saturday .. Sunday;
```

We can now take a `Day`, and see if we want to go to work:

```
Day : All_Days := Today;
if Day in Week_Days then
 go_to_work;
end if;
```

Or you could use the form **`if`** `Day` **`in range`** `Monday .. Friday` and we would not need the extra types.

Ada provides four useful attributes for enumeration type handling, note these are used slightly differently than many other attributes as they are applied to the type, not the object.

Succ This attribute supplies the 'successor' to the current value, so the '`Succ` value of an object containing `Monday` is `Tuesday`.
Note: If the value of the object is `Sunday` then an exception is raised, you cannot `Succ` past the end of the enumeration.

Pred This attribute provides the 'predecessor' of a given value, so the `'Pred` value of an object containing `Tuesday` is `Monday`.
Note: the rule above still applies `'Pred` of `Monday` is an error.

Val This gives you the value (as a member of the enumeration) of element n in the enumeration. Thus `Val(2)` is `Wednesday`.
Note: the rule above still applies, and note also that '`Val(0)` is the same as '`First`.

Pos This gives you the position in the enumeration of the given element name. Thus '`Pos(Wednesday)` is `2`.
Note: the range rules still apply, also that `'Last` will work, and return `Sunday`.

```
All_Days'Succ(Monday) = Tuesday
All_Days'Pred(Tuesday) = Monday
All_Days'Val(0) = Monday
All_Days'First = Monday
All_Days'Val(2) = Wednesday
All_Days'Last = Sunday
All_Days'Succ(All_Days'Pred(Tuesday)) = Tuesday
```

Ada also provides a set of 4 attributes for range types, these are intimatly associated with those above and are:

First This provides the value of the first item in a range. Considering the range `0 .. 100` then `'First` is `0`.

Last This provides the value of the last item in a range, and so considering above, `'Last` is `100`.

Length This provides the number of items in a range, so `'Length` is actually `101`.

Range This funnily enough returns in this case the value we gave it, but you will see when we come onto arrays how useful this feature is.

As you can see these have no direct C/C++ equivalent and are part of the reason for Ada's reputation for safety, you can define for a parameter exactly the range of values it might take, it all amounts to better practice for large developments where your interface is read by many people who may not be able to tell that the integer parameter day starts at 0, which indicates Wednesday etc.

1.1.6 Arrays *{3.6}*.

Arrays in Ada make use of the range syntax to define their bounds and can be arrays of any type, and can even be declared as unknown size.

Some example:

```
char name[31];
int track[3];
int dbla[3][10];
int init[3] = { 0, 1, 2 };
typedef char[31] name_type;
track[2] = 1;
dbla[0][3] = 2;
```

```
Name  : array (0 .. 30) of Character; -- OR
Name  : String (1 .. 30);
Track : array (0 .. 2) of Integer;
DblA  : array (0 .. 2) of array (0 .. 9) of Integer; -- OR
DblA  : array (0 .. 2,0 .. 9) of Integer;
Init  : array (0 .. 2) of Integer := (0, 1, 2);
type Name_Type is array (0 .. 30) of Character;
track(2)  := 1;
dbla(0,3) := 2;
-- Note try this in C.
a, b : Name_Type;
a := b; -- will copy all elements of b into a.
```

Simple isn't it, you can convert C arrays into Ada arrays very easily. What you don't get is all the things you can do with Ada arrays that you can't do in C/C++.

non-zero based ranges. Because Ada uses ranges to specify the bounds of an array then you can easily set the lower bound to anything you want, for example:

```
Example : array (-10 .. 10) of Integer;
```

non-integer ranges. In the examples above we have used the common abbreviation for range specifiers. The ranges above are all integer ranges, and so we did not need to use the correct form which is:

```
array(type range low .. high)
```

which would make Example above `array(Integer range -10 ..  10)`. Now you can see where we're going, take an enumerated type, `All_Days` and you can define an array:

```
Hours_Worked : array (All_Days range Monday .. Friday);
```

unbounded array types. The examples above did demonstrate how to declare an array type. One of Ada's goals is reuse, and to have to define a function to deal with a 1..10 array, and another for a 0..1000 array is silly. Therefore Ada allows you to define unbounded array types. An unbounded type can be used as a parameter type, but you cannot simply define a variable of such a type. Consider:

```
type  Vector is array (Integer range <>) of Float;
procedure sort_vector(sort_this : in out Vector);
Illegal_Variable : Vector;
Legal_Variable   : Vector(1..5);
subtype SmallVector is Vector(0..1);
Another_Legal    : SmallVector;
```

This does allow us great flexibility to define functions and procedures to work on arrays regardless of their size, so a call to `sort_vector` could take the `Legal_Variable` object or an object of type `SmallVector`, etc. *Note* that a variable of type `Smallvector` is constrained and so can be legally created.

array range attributes. If you are passed a type which is an unbounded array then if you want to loop through it then you need to know where it starts. So we can use the range attributes introduced in 1.1.5 to iterate over a given array thus: attributes for array types. Consider:

```
Example : array (1 .. 10) of Integer;
for i in Example'First .. Example'Last loop
for i in Example'Range loop
```

Note that if you have a multiple dimension array then the above notation implies that the returned values are for the first dimension, use the notation `Array_Name'attribute(dimension)` for multi-dimensional arrays.

Initialisation by range (Aggregates *{}*???) When initialising an array one can initialise a range of elements in one go:

```
Init : array (0 .. 3) of Integer := (0 .. 3 => 1);
Init : array (0 .. 3) of Integer := (0 => 1, others => 0);
```

The keyword **`others`** sets any elements not explicitly handled.

Slicing Array slicing is something usually done with memcpy in C/C++. Take a section out of one array and assign it into another.

```
Large : array (0 .. 100) of Integer;
Small : array (0 .. 3) of Integer;
-- extract section from one array into another.
Small(0 .. 3) := Large(10 .. 13);
-- swap top and bottom halfs of an array.
Large := Large(51 .. 100) & Large(1..50);
```

Note: Both sides of the assignment must be of the same type, that is the same dimensions with each element the same. The following is illegal.

```
-- extract section from one array into another.
Small(0 .. 3) := Large(10 .. 33);
--                      ^^^^^^^^ range too big.
```

1.1.7 Records *{3.8}*.

You shouldn't have too much problem here, you can see an almost direct mapping from C/C++ to Ada for simple structures. Note the example below does not try to convert type to type, thus the C char*, to hold a string is converted to the Ada String type.

```
struct _device {
 int major_number;
 int minor_number;
 char name[20];
};
typedef struct _device Device;

type struct_device is
 record
  major_number : Integer;
  minor_number : Integer;
  name : String(1 .. 19);
 end record;
type Device is new struct_device;
```

As you can see, the main difference is that the name we declare for the initial record is a type, and can be used from that point on. In C all we have declared is a structure name, we then require the additional step of typedef-ing to add a new type name.

Ada uses the same element reference syntax as C, so to access the minor_number element of an object lp1 of type Device we write `lp1.minor_number`. Ada does allow, like C, the initialisation of record members at declaration. In the code below we introduce a feature of Ada, the ability to name the elements we are going to initialise. This is useful for clarity of code, but more importantly it allows us to only initialise the bits we want.

```
Device lp1 = {1, 2, "lp1"};
lp1 : Device := (1, 2, "lp1");
lp2 : Device := (major_number => 1,
                 minor_number => 3,
                 name => "lp2");
tmp : Device := (major_number => 255,
                 name => "tmp");
```

When initialising a record we use an aggregate, a construct which groups together the members. This facility (unlike aggregates in C) can also be used to assign members at other times as well.

```
tmp : Device;
-- some processing
tmp := (major_number => 255, name => "tmp");
```

This syntax can be used anywhere where parameters are passed, initialisation (as above) function/procedure calls, variants and discriminants and generics. The code above is most useful if we have a default value for minor_number, so the fact that we left it out won't matter. This is possible in Ada.

This facility improves readability and as far as most Ada programmers believe maintainability.

```
type struct_device is
 record
  major_number : Integer := 0;
  minor_number : Integer := 0;
  name : String(1 .. 19) := "unknown";
 end record;
```

Structures/records like this are simple, and there isn't much more to say. The more interesting problem for Ada is modelling C unions (see section 1.1.10 on page 22).

1.1.8 Access types (pointers) *{3.10}*.

The topic of pointers/references/access types is the most difficult, each language has its own set of rules and tricks. In C/C++ the thing you must always remember is that the value of a pointer is the real memory address, in Ada it is not. It is a type used to access the data.

Ada access types are safer, and in some ways easier to use and understand, but they do mean that a lot of C code which uses pointers heavily will have to be reworked to use some other means.

The most common use of access types is in dynamic programming, for example in linked lists.

```
struct _device_event {
 int major_number;
 int minor_number;
 int event_ident;
 struct _device_event* next;
};
```

```
type Device_Event;
type Device_Event_Access is access Device_Event;
type Device_Event is
 record
  major_number : Integer := 0;
  minor_number : Integer := 0;
  event_ident  : Integer := 0;
  next : Device_Event_Access := null;
  -- Note: the assignement to null is not required,
  -- Ada automatically initialises access types to
  -- null if no other value is specified.
 end record;
```

The Ada code may look long-winded but it is also more expressive, the access type is declared before the record so a real type can be used for the declaration of the element next. *Note*: we have to forward declare the record before we can declare the access type, is this extra line worth all the moans we hear from the C/C++ community that Ada is overly verbose?

When it comes to dynamically allocating a new structure the Ada allocator syntax is much closer to C++ than to C.

```
Event_1 := new Device_Event;
Event_1.next := new Device_Event'(1, 2, EV_Paper_Low, null);
```

There are three things of note in the example above. Firstly the syntax, we can say directly that we want a new *thing*, none of this malloc rubbish. Secondly that there is no difference in syntax between access of elements of a statically allocated record and a dynamically allocated one. We use the `record.element` syntax for both. Lastly that we can initialise the values as we create the object, the tick is used again, not as an attribute, but with parenthases in order to form a qualified expresssion.

Ada allows you to assign between access types, and as you would expect it only changes what the access type points to, not the contents of what it points to. One thing to note again, Ada allows you to assign one structure to another if they are of the same type, and so a syntax is required to assign the contents of an access type, its easier to read than write, so:

```
dev1, dev2 : Device_Event;
pdv1, pdv2 : Device_Event_Access;
dev1 := dev2; -- all elements copied.
pdv1 := pdv2; -- pdv1 now points to contents of pdv2.
pdv1.all := pdv2.all; -- !!
```

What you may have noticed is that we have not discussed the operator to free the memory we have allocated, the equivalent of C's free() or C++'s delete.

There is a good reason for this, *Ada does not have one*.

To digress for a while, Ada was designed as a language to support garbage collection, that is the runtime would manage deallocation of no longer required dynamic memory. However at that time garbage collection was slow, required a large overhead in tracking dynamic memory and tended to make programs irratic in performance, slowing as the garbage collector kicks in. The language specification therefore states *{13.11}* "*An implementation need not support garbage collection* ...". This means that you must, as in C++ manage your own memory deallocation.

Ada requires you to use the generic procedure `Unchecked_Deallocation` (see 1.3.3 on page 36) to deallocate a dynamic object. This procedure must be instantiated for each dynamic type and should not (ideally) be declared on a public package spec, ie provide the client with a deallocation procedure which uses `Unchecked_Deallocation` internally.

1.1.9 Ada advanced types and tricks.

Casting (wow) As you might expect from what we have seen so far Ada must allow us some way to relax the strong typing it enforces. In C the cast allows us to make anything look like something else, in Ada type *coersion* can allow you to convert between two similar types, ie:

```
type Thing is new Integer;
an_Integer : Integer;
a_Thing : Thing;
an_Integer := a_Thing; -- illegal
an_Integer := Integer(a_Thing);
```

This can only be done between similar types, the compiler will not allow such coersion between very different types, for this you need the generic procedure `Unchecked_Conversion` (see 1.3.3 on page 35) which takes as an argument one type, and returns another. The only constraint on this is that they must be the same size.

Procedure types. ***{}*** Ada-83 did not allow the passing of procedures as subprogram parameters at execution time, or storing procedures in records etc. The rationale for this was that it broke the ability to statically prove the code. Ada-95 has introduced the ability to define types which are in effect similar to C's ability to define pointers to functions.

In C/C++ there is the most formidable syntax for defining pointers to functions and so the Ada syntax should come as a nice surprise:

```
typedef int (*callback_func)(int param1, int param2);

type Callback_Func is access function(param_1 : in Integer;
                                      param_2 : in Integer)
                                     return Integer;
```

Discriminant types *{3.7}*. Discriminant types are a way of parameterising a compound type (such as a record, tagged, task or protected type). For example:

```
type Event_Item is
 record
  Event_ID   : Integer;
  Event_Info : String(1 .. 80);
 end record;

type Event_Log(Max_Size : Integer) is
 record
  Log_Opened : Date_Type;
  Events : array (1 .. Max_Size) of Event_Item;
 end record;
```

First we declare a type to hold our event information in. We then declare a type which is a log of such events, this log has a maximum size, and rather than the C answer, define an array large enough for the maximum ever, or resort to dynamic programming the Ada approach is to instantiate the record with a max value and at time of instantiation define the size of the array.

```
My_Event_Log : Event_Log(1000);
```

If it is known that nearly all event logs are going to be a thousand items in size, then you could make that a default value, so that the following code is identical to that above.

```
type Event_Log(Max_Size : Integer := 1000) is
 record
  Log_Opened : Date_Type
  Events : array (Integer range 1 .. Max_Size) of Event_Item;
 end record;

My_Event_Log : Event_Log;
```

Again this is another way in which Ada helps, when defining an interface, to state precisely what we want to provide.

Variant records *{3.8.1}*. Anyone who has worked in a Pascal language will recognise variant records, they are a bit like C/C++ unions except that the are very different :-)

Ada variant records allow you to define a record which has 2 or more blocks of data of which only one is visible at any time. The visibility of the block is determined by a discriminant which is then ’cased’.

```
type Transport_Type is (Sports, Family, Van);

type Car(Type : Transport_Type) is
 record
  Registration_Date : Date_Type;
```

```
      Colour : Colour_Type;
      case Type is
       when Sports =>
        Soft_Top : Boolean;
       when Family =>
        Number_Seats : Integer;
        Rear_Belts : Boolean;
       when Van =>
        Cargo_Capacity: Integer;
      end case;
     end record;
```

So if you code `My_Car : Car(Family);` then you can ask for the number of seats in the car, and whether the car has seat belts in the rear, but you cannot ask if it is a soft top, or what its cargo capacity is.

I guess you've seen the difference between this and C unions. In a C union representation of the above any block is visible regardless of what type of car it is, you can easily ask for the cargo capacity of a sports car and C will use the bit pattern of the boolean to provide you with the cargo capacity. Not good.

To simplify things you can subtype the variant record with types which define the variant (note in the example the use of the designator for clarity).

```
    subtype Sports_Car is Car(Sports);
    subtype Family_Car is Car(Type => Family);
    subtype Small_Van  is Car(Type => Van);
```

Exceptions *{11.1}*. Exceptions are a feature which C++ is only now getting to grips with, although Ada was designed with exceptions included from the beginning. This does mean that Ada code will use exceptions more often than not, and certainly the standard library packages will raise a number of possible exceptions.

Unlike C++ where an exception is identified by its type in Ada they are uniquely identified by name. To define an exception for use, simply

```
    parameter_out_of_range : Exception;
```

These look and feel like constants, you cannot assign to them etc, you can only raise an exception and handle an exception.

Exceptions can be argued to be a vital part of the safety of Ada code, they cannot easily be ignored, and can halt a system quickly if something goes wrong, far faster than a returned error code which in most cases is completely ignored.

System Representation of types *{13}*. As you might expect with Ada's background in embedded and systems programming there are ways in which you can force a type into specific system representations.

```
type BYTE is range 0 .. 255;
for BYTE use 8;
```

This first example shows the most common form of system representation clause, the size attribute. We have asked the compiler to give us a range, from 0 to 255 and the compiler is at liberty to provide the best type available to hold the representation. We are forcing this type to be 8 bits in size.

```
type DEV_Activity is (READING, WRITING, IDLE);
for DEV_Activity use (READING => 1, WRITING => 2, IDLE => 3);
```

Again this is useful for system programming it gives us the safety of enumeration range checking, so we can only put the correct value into a variable, but does allow us to define what the values are if they are being used in a call which expects specific values.

```
type DEV_Available is BYTE;
for DEV_Available use at 16#00000340#;
```

This example means that all objects of type `DEV_Available` are placed at memory address 340 (Hex). This placing of data items can be done on a per object basis by using:

```
type DEV_Available is BYTE;
Avail_Flag : DEV_Available;
for Avail_Flag'Address use 16#00000340#;
```

Note the address used Ada's version of the C 0x340 notation, however the general form is `base#number#` where the base can be anything, including 2, so bit masks are real easy to define, for example:

```
Is_Available : constant BYTE := 2#1000_0000#;
Not_Available: constant BYTE := 2#0000_0000#;
```

Another feature of Ada is that any underscores in numeric constants are ignored, so you can break apart large numbers for readability.

```
type DEV_Status is 0 .. 15;

type DeviceDetails is
 record
  status : DEV_Activity;
  rd_stat: DEV_Status;
  wr_stat: DEV_Status;
 end record;

for DeviceDetails use
 record at mod 2;
  status  at 0 range 0 .. 7;
  rd_stat at 1 range 0 .. 3;
  wr_stat at 1 range 4 .. 7;
 end record;
```

This last example is the most complex, it defines a simple range type, and a structure. It then defines two things to the compiler, first the mod clause sets the byte packing for the structure, in this case back on two-byte boundaries. The second part of this structure defines exactly the memory image of the record and where each element occurs. The number after the 'at' is the byte offset and the range, or size, is specified in number of bits.

From this you can see that the whole structure is stored in two bytes where the first byte is stored as expected, but the second and third elements of the record share the second byte, low nibble and high nibble.

This form becomes very important a little later on.

1.1.10 C Unions in Ada, (food for thought).

Ada has more than one way in which it can represent a union as defined in a C program, the method you choose depends on the meaning and usage of the C union.

Firstly we must look at the two ways unions are identified. Unions are used to represent the data in memory in more than one way, the programmer must know which way is relevant at any point in time. This variant identifier can be inside the union or outside, for example:

```
struct _device_input {
 int device_id;
 union {
  type_1_data from_type_1;
  type_2_data from_type_2;
 } device_data;
};
void get_data_func(_device_input* from_device);

union device_data {
 type_1_data from_type_1;
 type_2_data from_type_2;
};
void get_data_func(int *device_id, device_data* from_device);
```

In the first example all the data required is in the structure, we call the function and get back a structure which holds the union and the identifier which denotes which element of the union is active. In the second example only the union is returned and the identifier is seperate.

The next step is to decide whether, when converting such code to Ada, you wish to maintain simply the concept of the union, or whether you are required to maintain the memory layout also. *Note*: the second choice is usually only if your Ada code is to pass such a structure to a C program or get one from it.

If you are simply retaining the concept of the union then you would **not** use the second form, use the first form and use a variant record.

```
type Device_ID is new Integer;
type Device_Input(From_Device : Device_ID) is
 record
```

```
   case From_Device is
    when 1 =>
     From_Type_1 : Type_1_Data;
    when 2 =>
     From_Type_2 : Type_2_Data;
   end case;
  end record;
```

The above code is conceptually the same as the first piece of C code, however it will probably look very different, you could use the following representation clause to make it look like the C code (type sizes are not important).

```
for Device_Input use
 record
  From_Device at 0 range 0 .. 15;
  From_Type_1 at 2 range 0 .. 15;
  From_Type_2 at 2 range 0 .. 31;
 end record;
```

You should be able to pass this to and from C code now. You could use a representation clause for the second C case above, but unless you really must pass it to some C code then re-code it as a variant record.

We can also use the abilities of `Unchecked_Conversion` to convert between different types (see 1.3.3 on page 35). This allows us to write the following:

```
type Type_1_Data is
 record
  Data_1 : Integer;
 end record;
type Type_2_Data is
 record
  Data_1 : Integer;
 end record;
function Type_1_to_2 is new Unchecked_Conversion
  (Source => Type_1_data, Target => Type_2_Data);
```

This means that we can read/write items of type `Type_1_Data` and when we need to represent the data as `Type_2_Data` we can simply write

```
Type_1_Object : Type_1_Data := ReadData;
:
Type_2_Object : Type_2_Data := Type_1_to_2(Type_1_Object);
```

1.2 C/C++ statements to Ada.

I present below the set of C/C++ statement types available, with each its Ada equivalent.

Note: All Ada statements can be qualified by a name, this be discussed further in the section on Ada looping constructs, however it can be used anywhere to improve readability, for example:

```
begin
 Init_Code:
  begin
   Some_Code;
  end Init_Code;
 Main_Loop:
  loop
   if Some_Value then
    exit loop Main_Loop;
   end if;
  end loop Main_Loop;
 Term_Code:
  begin
   Some_Code;
  end Term_Code;
end A_Block;
```

1.2.1 Compound Statement *{5.6}*

A compound statement is also known as a block and in C allows you to define variables local to that block, in C++ variables can be defined anywhere. In Ada they must be declared as part of the block, but must appear in the declare part just before the block starts.

```
{
 declarations
 statements
}
```

```
declare
 declarations
begin
 statement
end;
```

1.2.2 if Statement *{5.3}*

If statements are the primary selection tool available to programmers. The Ada if statement also has the ’elsif’ construct (which can be used more than once in any if statement), very useful for large complex selections where a switch/case statement is not possible.

Note: Ada does not require brackets around the expressions used in if, case or loop statements.

```
if (expression)
{
 statement
} else {
 statement
}
```

```
if expression then
 statement
elsif expression then
 statement
else
 statement
end if;
```

1.2.3 switch Statement *{5.4}*

The switch or case statement is a very useful tool where the number of possible values is large, and the selection expression is of a constant scalar type.

```
switch (expression)
{
 case value: statement
 default: statement
}
```

```
case expression is
 when value  => statement
 when others => statement
end case;
```

There is a point worth noting here. In C the end of the statement block between case statements is a break statement, otherwise we drop through into the next case. In Ada this does not happen, the end of the statement is the next case.

This leads to a slight problem, it is not uncommon to find a switch statement in C which looks like this:

```
switch (integer_value) {
case 1:
case 2:
case 3:
case 4:
  value_ok = 1;
  break;
case 5:
case 6:
case 7:
  break;
}
```

This uses ranges (see 1.1.5 on page 11) to select a set of values for a single operation, Ada also allows you to or values together, consider the following:

```
case integer_value is
 when 1 .. 4 => value_ok := 1;
 when 5 | 6 | 7 => null;
end case;
```

You will also note that in Ada there must be a statement for each case, so we have to use the Ada **`null`** statement as the target of the second selection.

1.2.4 Ada loops *{5.5}*

All Ada loops are built around the simple **`loop`** `...` **`end`** construct

```
loop
 statement
end loop;
```

1.2.4.1 while Loop

The while loop is common in code and has a very direct Ada equivalent.

```
while (expression)
{
 statement
}
```

```
while expression loop
 statement
end loop;
```

1.2.4.2 do Loop

The do loop has no direct Ada equivalent, though section 1.2.4.4 will show you how to synthesize one.

```
do
{
 statement
} while (expression)
```

```
-- no direct Ada equivalent.
```

1.2.4.3 for Loop

The for loop is another favourite, Ada has no direct equivalent to the C/C++ for loop (the most frighteningly overloaded statement in almost any language) but does allow you to iterate over a range, allowing you access to the most common usage of the for loop, iterating over an array.

```
for (init-statement ; expression-1 ; loop-statement)
{
 statement
}
```

```
for ident in range loop
 statement
end loop;
```

However Ada adds some nice touches to this simple statement.

Firstly, the variable ident is actually declared by its appearance in the loop, it is a new variable which exists for the scope of the loop only and takes the correct type according to the specified range.

Secondly you will have noticed that to loop for 1 to 10 you can write the following Ada code:

```
for i in 1 .. 10 loop
 null;
end loop;
```

What if you want to loop from 10 down to 1? In Ada you cannot specify a range of `10 .. 1` as this is defined as a ’null range’. Passing a null range to a for loop causes it to exit immediatly. The code to iterate over a null range such as this is:

```
for i in reverse 1 .. 10 loop
 null;
end loop;
```

1.2.4.4 break and continue

In C and C++ we have two useful statements break and continue which may be used to add fine control to loops. Consider the following C code:

```
while (expression) {
 if (expression1) {
  continue;
 }
 if (expression2) {
  break;
 }
}
```

This code shows how break and continue are used, you have a loop which takes an expression to determine general termination procedure. Now let us assume that during execution of the loop you decide that you have completed what you wanted to do and may leave the loop early, the break forces a ’jump’ to the next statement after the closing brace of the loop. A continue is similar but it takes you to the first statement after the opening brace of the loop, in effect it allows you to reevaluate the loop.

In Ada there is no continue, and break is now exit.

```
while expression loop
 if expression2 then
  exit;
 end if;
end loop;
```

The Ada exit statement however can combine the expression used to decide that it is required, and so the code below is often found.

```
while expression loop
 exit when expression2;
end loop;
```

This leads us onto the do loop, which can now be coded as:

```
loop
 statement
 exit when expression;
end loop;
```

Another useful feature which C and C++ lack is the ability to 'break' out of nested loops, consider

```
while ((!feof(file_handle) && (!percent_found)) {
 for (char_index = 0; buffer[char_index] != '\n'; char_index++) {
  if (buffer[char_index] == '%') {
   percent_found = 1;
   break;
  }
  // some other code, including get next line.
 }
}
```

This sort of code is quite common, an inner loop spots the termination condition and has to signal this back to the outer loop. Now consider

```
Main_Loop:
while not End_Of_File(File_Handle) loop
 for Char_Index in Buffer'Range loop
  exit when Buffer(Char_Index) = NEW_LINE;
  exit Main_Loop when Buffer(Char_Index) = PERCENT;
 end loop;
end loop Main_Loop;
```

1.2.5 return *{6.5}*

Here again a direct Ada equivalent, you want to return a value, then return a value,

```
return value; // C++ return

return value; -- Ada return
```

1.2.6 labels and goto *{5.8}*

Don't do it !!, OK one day you might need to, so heres how. Declare a label and jump to it.

```
label:
 goto label;
```

```
<<label>>
 goto label;
```

1.2.7 exception handling *{11.2}*

Ada and the newer verions of C++ support exception handling for critical errors. Exception handling consists of three components, the exception, raising the exception and handling the exception.

In C++ there is no exception type, when you raise an exception you pass out any sort of type, and selection of the exception is done on its type. In Ada as seen above there is a 'psuedo-type' for exceptions and they are then selected by name.

Firstly lets see how you catch an exception, the code below shows the basic structure used to protect statement1, and execute statement2 on detection of the specified exception.

```
try {
 statement1
} catch (declaration) {
 statement2
}
```

```
begin
 statement1
exception
 when ident => statement2
 when others => statement2
end;
```

Let us now consider an example, we will call a function which we know may raise a particular exception, but it may raise some we don't know about, so we must pass anything else back up to whoever called us.

```
try {
 function_call();
} catch (const char* string_exception) {
 if (!strcmp(string_exception, "the_one_we_want")) {
  handle_it();
 } else {
  throw;
 }
} catch (...) {
 throw;
}

begin
 function_call;
exception
 when the_one_we_want => handle_it;
 when others => raise;
end;
```

This shows how much safer the Ada version is, we know exactly what we are waiting for and can immediately process it. In the C++ case all we know is that an exception of type 'const char*' has been raised, we must then check it still further before we can handle it.

You will also notice the similarity between the Ada exception catching code and the Ada case statement, this also extends to the fact that the when statement can catch multiple exceptions. Ranges of exceptions are not possible, however you can or exceptions, to get:

```
begin
 function_call;
exception
 when the_one_we_want |
      another_possibility => handle_it;
 when others => raise;
end;
```

This also shows the basic form for raising an exception, the throw statement in C++ and the raise statement in Ada. Both normally raise a given exception, but both when invoked with no exception reraise the last one. To raise the exception above consider:

```
throw (const char*)"the_one_we_want";

raise the_one_we_want;
```

1.2.8 sub-programs

The following piece of code shows how C/C++ and Ada both declare and define a function. Declaration is the process of telling everyone that the function exists and what its type and parameters are. The definitions are where you actually write out the function itself. (In Ada terms the function spec and function body).

```
return_type func_name(parameters);
return_type func_name(parameters)
{
 declarations
 statement
}

function func_name(parameters) return return_type;
function func_name(parameters) return return_type is
```

```
     declarations
    begin
     statement
    end func_name;
```

Let us now consider a special kind of function, one which does not return a value. In C/C++ this is represented as a return type of void, in Ada this is called a procedure.

```
    void func_name(parameters);
    procedure func_name(parameters);
```

Next we must consider how we pass arguments to functions.

```
    void func1(int  by_value);
    void func2(int* by_address);
    void func3(int& by_reference); // C++ only.
```

These type of parameters are I hope well understood by C and C++ programmers, their direct Ada equivalents are:

```
    type int      is new Integer;
    type int_star is access int;
    procedure func1(by_value     : in     int);
    procedure func2(by_address   : in out int_star);
    procedure func3(by_reference : in out int);
```

Finally a procedure or function which takes no parameters can be written in two ways in C/C++, though only one is Ada.

```
    void func_name();
    void func_name(void);
    int  func_name(void);

    procedure func_name;
    function  func_name return Integer;
```

Ada also provides two features which will be understood by C++ programmers, possibly not by C programmers, and a third I don't know how C does without:

Overloading Ada allows more than one function/procedure with the same name as long as they can be uniquely identified by their signature (a combination of their parameter and return types).

```
    function Day return All_Days;
    function Day(a_date : in Date_Type) return All_Days;
```

The first returns you the day of week, of today, the second the day of week from a given date. They are both allowed, and both visible. The compiler decides which one to use by looking at the types given to it when you call it.

Operator overloading *{6.6}* As in C++ you can redefine the standard operators in Ada, unlike C++ you can do this outside a class, and for any operator, with any types. The syntax for this is to replace the name of the function (operators are always functions) with the operator name in quotes, ie:

```
function "+"(Left, Right : in Integer) return Integer;
```

Available operators are:

=	<	<=	>=	>
+	-	&	**abs**	**not**
*	/	**mod**	**rem**	**
and	**or**	**xor**		

Parameter passing modes C++ allows three parameter passing modes, by value, by pointer and by reference (the default mode for Ada).

```
void func(int by_value, int* by_pointer, int& by_reference);
```

Ada provides two optional keywords to specify how parameters are passed, **`in`** and **`out`**. These are used like this:

```
procedure proc(Parameter : in     Integer);
procedure proc(Parameter :    out Integer);
procedure proc(Parameter : in out Integer);
procedure proc(Parameter :        Integer);
```

If these keywords are used then the compiler can protect you even more, so if you have an **`out`** parameter it will warn you if you use it before it has been set, also it will warn you if you assign to an **`in`** parameter.

Note that you cannot mark parameters with **`out`** in functions as functions are used to return values, such *side affects* are disallowed.

Default parameters *{6.4.1}* Ada (and C++) allow you to declare default values for parameters, this means that when you call the function you can leave such a parameter off the call as the compiler knows what value to use.

```
procedure Create
 (File : in out File_Type;
  Mode : in     File_Mode := Inout_File;
  Name : in     String := "";
  Form : in     String := "");
```

This example is to be found in each of the Ada file based IO packages, it opens a file, given the file 'handle' the mode, name of the file and a system independant 'form' for the file. You can see that the simplest invokation of Create is `Create(File_Handle);` which simply provides the handle and all other parameters are defaulted (In the Ada library a file name of "" implies opening a temporary file). Now suppose that we wish to provide the name of the file also, we would have to write `Create(File_Handle, Inout_File, "text.file");` wouldn't we? The Ada answer is no. By using designators as has been demonstrated above we could use the form:

```
    Create(File => File_Handle,
           Name => "text.file");
```

and we can leave the mode to pick up its default. This skipping of parameters is a uniquely Ada feature.

Nested procedures Simple, you can define any number of procedures within the definition of another as long as they appear before the begin.

```
    procedure Sort(Sort_This : in out An_Array) is
     procedure Swap(Item_1, Item_2 : in out Array_Type) is
     begin
     end Swap;
    begin
    end Sort;
```

Notes: you can get in a mess with both C++ and Ada when mixing overloading and defaults. For example:

```
    procedure increment(A_Value : A_Type);
    procedure increment
      (A_Value : in out A_Type;
       By      : in     Integer := 1);
```

If we call increment with one parameter which of the two above is called? Now the compiler will show such things up, but it does mean you have to think carefully and make sure you use defaults carefully.

1.3 Ada Safety.

Ada is probably best known for its role in safetly critical systems. Ada is probably best known for its role in safety critical systems. Boeing standardized on Ada as the language for the new 777, and I can assure you such a decision is not taken lightly.

Ada is also commonly assumed to be a military language, with the US Department of Defense its prime advocate, this is not the case, a number of commercial and government developments have now been implemented in Ada. Ada is an excellent choice if you wish to spend your development time solving your customers problems, not hunting bugs in C/C++ which an Ada compiler would not have allowed.

1.3.1 Static provability.

Ada-83 did not provide Object Oriented features, and did not even provide procedural types as such constructs meant that you could only follow the path of the code at runtime. Ada-83 was statically provable, you could follow the route the code would take given certain inputs from the source code alone. This has been a great benefit and has provided Ada programmers with a great deal of confidence in the code they wrote.

Ada-95 has introduced these new features, Object Oriented programming through tagged types and procedural types which make it more difficult to statically prove an Ada-95 program, but the language designers decided that such features merited their inclusion in the language to further another goal, that of high reuse.

1.3.2 Predefined exceptions and pragmas.

A number of exceptions can be raised by the standard library and/or the runtime environment. You may expect to come accross at least one while you are learning Ada (and more once you know it ;-).

`Constraint_Error`

This exception is raised when a constraint is exceeded, such constraints include

- Numeric under/overflow.
- Range bounds exceeded.
- Reference to invalid record component.
- Dereference of **`null`** access type.

`Program_Error`

This is raised by the run-time to mark an erroneous program event, such as calling a procedure before package initialisation, or bad instantiation of a generic package.

`Storage_Error`

This exception is raised when a call to **`new`** could not be satisfied due to lack of memory.

`Tasking_Error`

This is raised when problems occur during tasking rendezvous (see section 7.2 on page 58).

This is not a list of the predefined pragmas *{L}* what I have provided is the set of options to the pragma `Supress` which can be used to stop certain run-time checks taking place. The pragma works from that point to the end of the innermost enclosing scope, or the end of the scope of the named object (see below).

`Access_Check`

Raises `Constraint_Error` on dereference of a **`null`** access value.

`Accessibility_Check`

Raises `Program_Error` on access to inaccessible object or subprogram.

`Discriminant_Check`

Raises `Constraint_Error` on access to incorrect component in a discriminant record.

`Division_Check`

Raises Constraint_Error on divide by zero.

`Elaboration_Check`

Raises Program_Error on unelaborated package or subprogram body.

`Index_Check`

Raises `Constraint_Error` on out of range array index.

`Length_Check`

Raises `Constraint_Error` on array length violation.

`Overflow_Check`

Raises `Constraint_Error` on overflow from numeric operation.

`Range_Check`

Raises `Constraint_Error` on out of range scalar value.

`Storage_Check`

Raises `Storage_Error` if not enough storage to satisfy a **`new`** call.

`Tag_Check`

Raises `Constraint_Error` if object has an invalid tag for operation.

```
pragma Suppress(Access_Check);
pragma Suppress(Access_Check, On => My_Type_Ptr);
```

The first use of the pragma above turns off checking for **`null`** access values throughout the code (for the lifetime of the suppress), whereas the second only suppresses the check for the named data item.

The point of this section is that by default *all* of these checks are enabled, and so any such errors will be trapped.

1.3.3 Unchecked programming.

You can subvert some of Adas type consistency by the use of unchecked programming. This is basically a set of procedures which do unsafe operations. These are:

`Unchecked_Conversion`

This generic function is defined as:

```
generic
type Source (<>) is limited private;
type Target (<>) is limited private;
function Ada.Unchecked_Conversion (Source_Object : Source)
                                                     return Target;
```

and should be instantiated like the example below (taken from one of the Ada-95 standard library packages `Ada.Interfaces.C`).

```
function Character_To_char is new
  Unchecked_Conversion (Character, char);
```

and can then be used to convert and Ada character to a C char, thus

```
A_Char : Interfaces.C.char := Character_To_char('a');
```

`Unchecked_Deallocation`

This generic function is defined as:

```
generic
type Object (<>) is limited private;
type Name is access Object;
procedure Ada.Unchecked_Deallocation (X : in out Name);
```

this function, instantiated with two parameters, only requires one for operation,

```
type My_Type is new Integer;
type My_Ptr  is access My_Type;
procedure Free is new Unchecked_Deallocation (My_Type, My_Ptr);
Thing : My_Ptr := new My_Type;
Free(Thing);
```

2

Ada Packages. *{7}*

Ada has one feature which many C/C++ programmers like to think they have an equivalent too - the package - they do not.

It is worth first looking at the role of header files in C/C++. Header files are simply program text which by virtue of the preprocessor are inserted into the compilers input stream. The `#include` directive knows nothing about what it is including and can lead to all sorts of problems, such as people who `#include "thing.c"`. This sharing of code by the preprocessor lead to the `#ifdef` construct as you would have different interfaces for different people. The other problem is that C/C++ compilations can sometime take forever because a included b included c ... or the near fatal a included a included a ...

Stroustrup has tried ref [9] (in vain, as far as I can see) to convince C++ programmers to remove dependance on the preprocessor but all the drawbacks are still there.

Any Ada package on the other hand consists of two parts, the specification (header) and body (code). The specification however is a completely stand alone entity which can be compiled on its own and so must include specifications from other packages to do so. An Ada package body at compile time must refer to its package specification to ensure legal declarations, but in many Ada environments it would look up a compiled version of the specification.

The specification contains an explicit list of the visible components of a package and so there can be no *internal knowledge* exploited as is often the case in C code, ie module a contains a functions aa() but does not export it through a header file, module b knows how a is coded and so uses the `extern` keyword to declare knowledge of it, and use it. C/C++ programmers therefore have to mark private functions and data as `static`.

2.1 What a package looks like

Below is the skeleton of a package, spec and body.

```
--file example.ads, the package specification.
package example is
:
:
end example;
```

```
--file example.adb, the package body.
package body example is
:
:
end example;
```

2.2 Include a package in another

Whereas a C file includes a header by simply inserting the text of the header into the current compilation stream with `#include "example.h"`, the Ada package specification has a two stage process.

Working with the example package above let us assume that we need to include another package, say `My_Specs` into this package so that it may be used. Firstly where do you insert it? Like C, package specifications can be inserted into either a specification or body depending on who is the client. Like a C header/code relationship any package included in the specification of package A is visible to the body of A, but not to clients of A. Each package is a seperate entity.

```
-- Specification for package example
with Project_Specs;
package example is
 type My_Type is new Project_Spec.Their_Type;
end example;

-- Body for package example
with My_Specs;
package body example is
 type New_Type_1 is new My_Specs.Type_1;
 type New_Type_2 is new Project_Specs.Type_1;
end example;
```

You can see here the basic visibility rules, the specification has to include `Project_Specs` so that it can declare `My_Type`. The body automatically inherits any packages included in its spec, so that you can see that although the body does not include `Project_Specs` that package is used in the declaration of `New_Type_1`. The body also includes another package `My_Specs` to declare the new type `New_Type_2`, the specification is unaware of this include and so cannot use `My_Specs` to declare new types. In a similar way an ordinary client of the package `example` cannot use the inclusion of `Project_Specs`, they would have to include it themselves.

To use an item, say a the type `Type_1` you must name it `My_Specs.Type_1`, in effect you have included the package name, not its contents. To get the same effect as the C `#include` you must also add another statement to make:

```
with My_Specs; use My_Specs
package body example is
:
:
end example;
```

It is usual in Ada to put the with and the use on the same line, for clarity. There is much more to be said about Ada packages, but that should be enough to start with. There is a special form of the **`use`** statement which can simply include an element (types only) from a package, consider:

```
use type Ada.Calendar.Time;
```

2.3 Package data hiding *{7.3}*

Data encapulation requires, for any level of safe reuse, a level of hiding. That is to say we need to defer the declaration of some data to a future point so that any client cannot depend on the structure of the data and allows the provider the ability to change that structure if the need arises.

In C this is done by presenting the 'private type' as a `void`* which means that you cannot know anything about it, but implies that no one can do any form of type checking on it. In C++ we can forward declare classes and so provide an anonymous class type.

```
/* C code */
typedef void* list;
list create(void);
// C++
class Our_List {
public:
 Our_List(void);
private:
 class List_Rep;
 List_Rep* Representation;
};
```

You can see that as a C++ programmer you have the advantage that when writing the implementation of `Our_List` and its internal representation `List_Rep` you have all the advantages of type checking, but the client still knows absolutely nothing about how the list is structured.

In Ada this concept is formalised into the 'private part' of a package. This private part is used to define items which are forward declared as private.

```
package Our_List is
type List_Rep is private;
function Create return List_Rep;
private
 type List_Rep is
  record
   -- some data
  end record;
end Our_List;
```

As you can see the way the Ada private part is usually used the representation of `List_Rep` is exposed, but because it is a private type the only operations that the client may use are = and /=, all other operations must be provided by functions and procedures in the package.

Note: we can even restrict use of = and /= by declaring the type as **`limited private`** when you wish to have no predefined operators available.

You may not in the public part of the package specification declare variables of the private type as the representation is not yet known, we can declare constants of the type, but you must declare them in both places, forward reference them in the public part with no value, and then again in the private part to provide a value:

```
package Example is
 type A is private;
 B : constant A;
private
 type A is new Integer;
 B : constant A := 0;
end Example;
```

To get exactly the same result as the C++ code above then you must go one step further, you must not expose the representation of `List_Rep`, and so you might use:

```
package Our_List is
 type List_Access is limited private;
 function Create return List_Access;
private
 type List_Rep; -- opaque type
 type List_Access is access List_Rep;
end Our_List;
```

We now pass back to the client an access type, which points to a ’deferred incomplete type’ whose representation is only required to be exposed in the package body.

2.4 Hierarchical packages.

Ada allows the nesting of packages within each other, this can be useful for a number of reasons. With Ada-83 this was possible by nesting package specs and bodies physically, thus:

```
package Outer is
 package Inner_1 is
 end Inner_1;
 package Inner_2 is
 end Inner_2;
private
end Outer;
```

Ada-95 has added to this the possibility to define child packages outside the physical scope of a package, thus:

```
package Outer is
package Inner_1 is
end Inner_1;
end Outer;
package Outer.Inner_2 is
end Outer.Inner_2;
```

As you can see `Inner_2` is still a child of outer but can be created at some later date, by a different team.

2.5 Renaming identifiers.

This is not a package specific topic, and it is only introduced here as the using of packages is the most common place to find a renames clause.

Consider:

```
with Outer;
with Outer.Inner_1;
package New_Package is
 OI_1 renames Outer.Inner_1;
 type New_type is new OI_1.A_Type;
end New_Package;
```

The use of `OI_1` not only saves us a lot of typing, but if outer were the package `Sorting_Algorithms`, and `Inner_1` was `Insertion_Sort`, then we could have
`Sort renames Sorting_Algorithms.Insertion_Sort`
and then at some later date if you decide that a quick sort is more approriate then you simply change the renames clause, and the rest of the package spec stays exactly the same.

Similarly if you want to include 2 functions from two different package with the same name then, rather than relying on overloading, or to clarify your code text you could:

```
with Package1;
function Function1 return Integer renames Package1.Function;
with Package2;
function Function2 return Integer renames Package2.Function;
```

Another example of a renames clause is where you are using some complex structure and you want to in effect use a synonym for it during some processing. In the example below we have a device handler structure which contains some procedure types which we need to execute in turn. The first example contains a lot of text which we don't really care about, so the second removes most of it, thus leaving bare the real work we are attempting to do.

```
for device in Device_Map loop
 Device_Map(device).Device_Handler.Request_Device;
 Device_Map(device).Device_Handler.
                     Process_Function(Process_This_Request);
```

```
  Device_Map(device).Device_Handler.Relinquish_Device;
end loop;

for device in Device_Map loop
  declare
    Device_Handler : Device_Type renames
    Device_Map(device).Device_Handler;
  begin
    Device_Handler.Request_Device;
    Device_Handler.Process_Function(Process_This_Request);
    Device_Handler.Relinquish_Device;
  end;
end loop;
```

3

Ada-95 Object Oriented Programming.

C++ extends C with the concept of a `class`. A class is an extension of the existing `struct` construct which we have reviewed in section 1.1.7 on page 15. The difference with a class is that a class not only contains data (member attributes) but code as well (member functions). A class might look like:

```
class A_Device {
public:
 A_Device(char*, int, int);
 char* Name(void);
 int   Major(void);
 int   Minor(void);
protected:
 char* name;
 int   major;
 int   minor;
};
```

This defines a class called A_Device, which encapsulates a Unix-like /dev entry. Such an entry has a name and a major and minor number, the actual data items are protected so a client cannot alter them, but the client can see them by calling the public interface functions.

The code above also introduces a constructor, a function with the same name as the class which is called whenever the class is created. In C++ these may be overloaded and are called either by the `new` operator, or in local variable declarations as below.

```
A_Device lp1("lp1", 10, 1);
A_Device* lp1;
  lp1 = new A_Device("lp1", 10, 1);
```

Creates a new device object called `lp1` and sets up the name and major/minor numbers.

Ada has also extended its equivalent of a struct, the `record` but does not directly attach the member functions to it. First the Ada equivalent of the above class is

```
package Devices is
 type Device is tagged private;
```

```
     type Device_Type is access Device;
     function Create(Name  : String;
                     Major : Integer;
                     Minor : Integer)   return Device_Type;
     function Name(this : Device_Type)  return String;
     function Major(this : Device_Type) return Integer;
     function Minor(this : Device_Type) return Integer;

    private
     type Device is tagged
      record
       Name  : String(1 .. 20);
       Major : Integer;
       Minor : Integer;
      end record;
    end Devices;
```

and the equivalent declaration of an object would be:

```
    lp1 : Devices.Device_Type := Devices.Create("lp1", 10, 1);
```

3.1 The tagged type.

The addition of the keyword **tagged** to the definition of the type Device makes it a class in C++ terms. The tagged type is simply an extension of the Ada-83 record type but (in the same way C++'s `class` is an extension of C's `struct`) which includes a 'tag' which can identify not only its own type but its place in the type hierarchy.

The tag can be accessed by the attribute `'Tag` but should only be used for comparison, ie

```
    dev1, dev2 : Device;
    if dev1'Tag = dev2'Tag then
```

this can identify the isa relationship between two objects.

Another important attribute `'Class` exists which is used in type declarations to denote the *class-wide type*, the inheritence tree rooted at that type, ie

```
    type Device_Class is Device'Class;
    -- or more normally
    type Device_Class is access Device'Class;
```

The second type denotes a pointer to objects of type `Device` and any objects whos type has been inherited from `Device`.

3.2 Class member attributes.

Member attributes in C++ directly map onto data members of the tagged type. So the `char* name` directly maps into `Name : String`.

3.3 Class member functions.

Non-virtual, non-const, non-static member functions map onto subprograms, within the same package as the tagged type, whos first parameter is of that tagged type or an access to the tagged type, or who returns such a type.

3.4 Virtual member functions.

Virtual member functions map onto subprograms, within the same package as the tagged type, whos first parameter is of the class-wide type, or an access to the class-wide type, or who returns such a type.

A pure virtual function maps onto a virtual member function with the keywords **`is abstract`** before the semicolon. When any pure virtual member functions exist the tagged type they refer to must also be identified as abstract. Also, if an abstract tagged type has been introduced which has no data, then the following shorthand can be used:

```
    type Root_Type is abstract tagged null record;
```

3.5 Static members.

Static members map onto subprograms within the same package as the tagged type. These are no different from normal Ada-83 subprograms, it is up to the programmer when applying coding rules to identify only member functions or static functions in a package which includes a tagged type.

3.6 Constructors/Destructors for Ada.

As you can see from the example above there is no constructors and destructors in Ada. In the example above we have synthesised this with the `Create` function which creates a new object and returns it. If you intend to use this method then the most important thing to remember is to use the same name throughout, `Create Copy Destroy` etc are all useful conventions.

Ada does provide a library package `Ada.Finalization` which can provide constructor/destructor like facilities for tagged types.

Note: See [4].

3.7 Inheritance, single and multiple.

The most common attribute sited as the mark of a true object oriented language is support for inheritance. Ada-95 adds this as *tagged type extension*.

For example, let us now inherit the device type above to make a tape device, firstly in C++

```
class A_Tape : public A_Device {
public:
 A_Tape(char*, int, int);
 int Block_Size(void);
protected:
 int block_size;
};
```

Now let us look at the example in Ada.

```
package Device.Tapes is
 type Tape is new device with private;
 type Tape_Type is access Tape;
 function Create(Name : String;
                 Major : Integer;
                 Minor : Integer) return Tape_Type;
 function Block_Size(this : Tape_Type) return Integer;
private
 type Tape is new Device with
  record
   Block_Size : Integer;
  end record;
end Device.Tapes;
```

Ada does not directly support multiple inheritance, ref [5] has an example of how to synthesise mulitple inheritance.

3.8 public/protected/private.

In the example at the top of this section we provided the `Device` comparison. In this example the C++ class provided a public interface and a protected one, the Ada equivalent then provided an interface in the public part and the tagged type declaration in the private part. Because of the rules for child packages (see 2.4 on page 40) a child of the `Devices` package can see the private part and so can use the definition of the `Device` tagged type.

Top mimic C++ private interfaces you can choose to use the method above, which in effect makes them protected, or you can make them really private by using opaque types (see 2.3 on page 39).

3.9 A more complete example.

```
class base_device {
public:
 char* name(void) const;
 int   major(void) const;
 int   minor(void) const;
 enum { block, character, special } io_type;
 io_type type(void) const;
 char read(void) = 0;
 void write(char) = 0;
 static char* type_name(void);
protected:
 char* _name;
 int   _major;
 int   _minor;
 static const io_type _type;
 base_device(void);
private:
 int _device_count;
};
```

The class above shows off a number of C++ features,

- Some const member functions.
- Some pure virtual member functions.
- A Static member function.
- Some protected member attributes.
- A Static const member attribute.
- A protected constructor.
- A private member attribute.

All of these, including the reasons why they might be used should be familiar to you, below is an equivalent specification in Ada.

```
package Devices is
 type Device is abstract tagged limited private;
 type Device_Type  is access Device;
 type Device_Class is access Device'Class;
 type IO_Type is (Block, Char, Special);
 function Name(this   : in Device_Type) return String;
 function Major(this  : in Device_Type) return Integer;
 function Minor(this  : in Device_Type) return Integer;
 function IOType(this : in Device_Type) return IO_Type;
 function Read(this   : Device_Class)   return Character is abstract;
 procedure Write(this : Device_Class; Output : Character) is abstract;
 function Type_Name return String;
private
 type Device_Count;
 type Device_Private is access Device_Count;
 type Device is abstract tagged limited
 record
  Name  : String(1 .. 20);
```

```
       Major : Integer;
       Minor : Integer;
       Count : Device_Private;
      end record;
      Const_IO_Type   : constant IO_Type := special;
      Const_Type_Name : constant String := "Device";
     end Devices;
```

4 Generics

One of Ada's strongest claims is the ability to code for reuse. C++ also claims reuse as one of its goals through Object Oriented Programming. Ada-83 allowed you to manage the data encapsulation and layering through the package mechanism and Ada-95 does include proper facilities for OO Programming. Where Ada led however, and C++ is following is the area of generic, or template programming.

4.1 A generic procedure *{12.6}*

For example. A sort algorithm is well understood, and we may like to code a sort for an array of int's in C, we would have a function like:

```
void sort(int *array, int num_elements);
```

however when you come to sort an array of structures you either have to rewrite the function, or you end up with a generic sort function which looks like this:

```
void sort(void *array, int element_size, int element_count,
          int (*compare)(void* el1, void *el2));
```

This takes a bland address for the start of the array user supplied parameters for the size of each element and the number of elements and a function which compares two elements. C does not have strong typing, but you have just stripped away any help the compiler might be able to give you by using `void*`.

Now let us consider an Ada generic version of the sort function:

```
generic
 type index_type is (<>);
 type element_type is private;
 type element_array is array (index_type range <>) of element_type;
 with function "<" (el1, el2 : element_type) return Boolean;
procedure Sort(the_array : in out element_array);
```

This shows us quite a few features of Ada generics and is a nice place to start, for example note that we have specified a lot of detail about the thing we are going to sort, it is an array, for which we don't know the bounds so it is specified as `range` <>. We also can't expect that the range is an integer range and so we must also make the range type a parameter, `index_type`. Then we come onto the element type, this is simply specified as private, so all we know is that we can test equality and assign one to another. Now that we have specified exactly what it is we are going to sort we must ask for a function to compare two elements, similar to C we must ask the user to supply a function, however in this case we can ask for an operator function and notice that we use the keyword **`with`** before the function.

I think that you should be able to see the difference between the Ada code and C code as far as readability (and therefore maintainability) are concerned and why, therefore, Ada promotes the reuse philosophy.

Now let's use our generic to sort some of `MyTypes`.

```
MyArray : array (Integer 0 .. 100) of MyType;
function LessThan(el1, el2 : MyType) return Boolean;
procedure SortMyType is new Sort(Integer, MyType, MyArray, LessThan);
SortMyType(MyArray);
```

The first two lines simply declare the array we are going to sort and a little function which we use to compare two elements (*note*: no self respecting Ada programmer would define a function `LessThan` when they can use "<", this is simply for this example). We then go on to instantiate the generic procedure and declare that we have an array called `MyArray` of type `MyType` using an `Integer` range and we have a function to compare two elements. Now that the compiler has instantiated the generic we can simply call it using the new name.

Note: The Ada compiler instantiates the generic and will ensure type safety throughout.

4.2 Generic packages *{12.7}*

Ada packages and generics where designed to go together, you will even find generic packages in the Ada standard library. For example:

```
generic
type Element_Type is private;
package Ada.Direct_IO is
```

Is the standard method for writing out binary data structures, and so one could write out to a file:

```
type My_Struct is
record
...
end record;
package My_Struct_IO is new Ada.Direct_IO(My_Struct);
use My_Struct_IO;
Item : My_Struct;
File : My_Struct_IO;
...
My_Struct_IO.Write(File, Item);
```

Note: see section 5.2 on page 53 for a more detailed study of these packages and how they are used.

4.3 Generic types and other parameters *{12.4}*

The types you may specify for a generic subprogram or package are as follows:

type X **is private**

We can know nothing about the type, except that we may test for equality and we may assign one to another. If we add in the keyword **`limited`** then even these abilities are unavailable.

type X(<>) **is private**

Added for Ada-95, this is similar to the parameter above except that we can define data items in the body of our package of type X, this may be illegal if the type passed is unconstrained, ie `String`. Ada-95 does not allow the instantiation of generics with unconstrained types, unless you use this syntax in which case you cannot declare data items of this type.

type X **is** (<>)

The type is a discrete type, Integer, Character, Enumeration etc.

type X **is range** <>

The type indicates a range, ie `0 .. 100.`

type X **is mod** <>

The type is a modulus type of unknown size (Added for Ada-95).

type X **is digits** <>

The type is a floating point type.

type X **is delta** <>

The type is a fixed point type.

type X **is tagged private**

The type is a tagged type, ie an Ada-95 extensible record.

There is one final parameter which may be passed to a generic package, another generic package (Added for Ada-95).

```
with Generic_Tree;
generic
 with package A_Tree is new Generic_Tree(<>);
package Tree_Walker is
 -- some code.
end Tree_Walker;
```

This says that we have some package called Generic_Tree which is a generic package implementing a tree of generic items. We want to be able to walk any such tree and so we say that we have a new generic package which takes a parameter which must be an instantiated package. ie

```
package AST is new Generic_Tree(Syntax_Element);
package AST_Print is new Tree_Walker(AST);
```

5

IO

A common area for confusion is the Ada IO model, this has been shaped by the nature of the language itself and specifically the strong typing which has a direct impact on the model used to construct the IO libraries. If you stop and think about it briefly it is quite clear that with the typing rules we have introduced above you cannot write a function like the C `write()` which takes any old thing and puts it out to a file, how can you write a function which will take any parameter, even types which will be introduced after it has been completed. Ada-83 took a two pronged approach to IO, with the package `Text_IO` for simple, textual input output, and the packages `Sequential_IO` and `Direct_IO` which are generic packages for binary output of structured data.

The most common problem for C and C++ programmers is the lack of the printf family of IO functions. There is a good reason for their absence in Ada, the use in C of variable arguments, the '...' at the end of the printf function spec. Ada cannot support such a construct as the type of each parameter is unknown.

5.1 Ada.Text_IO

The common way to do console-like IO, similar to C's printf(), puts() and putchar() is to use the package `Ada.Text_IO`. This provides a set of overloaded functions called `Put` and `Get` to read and write to the screen or to simple text files. There are also functions to open and close such files, check end of file conditions and to do line and page management.

A simple program below uses `Text_IO` to print a message to the screen, including numerics! These are achieved by using the types attribute `'Image` which gives back a String representation of a value.

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Test_IO is
begin
 Put_Line("Test Starts Here >");
 Put_Line("Integer is " & Integer'Image(2));
 Put_Line("Float is " & Float'Image(2.0));
 Put_Line("Test Ends Here");
end Test_IO;
```

It is also possible to use one of the generic child packages of `Ada.Text_IO` such as `Ada.Text_IO.Integer_IO` which can be instantiated with a particular type to provide type safe textual IO.

```
with Ada.Text_IO;
type My_Integer is new Integer;
package My_Integer_IO is new Ada.Text_IO.Integer_IO(My_Integer);
use My_Integer_IO;
```

5.2 Ada.Sequential_IO and Ada.Direct_IO

These two generic packages provide IO facilities for files which contain identical records. They can be instantiated in a similar way to the generic text IO packages above, so for example:

```
with Ada.Direct_IO;
package A_Database is
 type File_Header is
  record
   Magic_Number      : Special_Stamp;
   Number_Of_Records : Record_Number;
   First_Deleted     : Record_Number;
  end record;

 type Row is
  record
   Key  : String(1 .. 80);
   Data : String(1 .. 255);
  end record;
 package Header_IO is new Direct_IO (File_Header); use Header_IO;
 package Row_IO    is new Direct_IO (Row);         use Record_IO;
end A_Database;
```

Now that we have some instantiated packages we can read and write records and headers to and from a file. However we want each database file to consist of a header followed by a number of rows, so we try the following

```
declare
 Handle   : Header_IO.File_Type;
 A_Header : File_Header;
 A_Row    : Row;
begin
 Header_IO.Open(File => Handle, Name => "Test");
 Header_IO.Write(Handle, A_Header);
 Row_IO.Write(Handle, A_Row);
 Header_IO.Close(Handle);
end;
```

The obvious error is that `Handle` is defined as a type exported from the `Header_IO` package and so cannot be passed to the procedure `Write` from the package `Row_IO`. This strong typing means that both `Sequential_IO` and `Direct_IO` are designed only to work on files containg all elements of the same type.

When designing a package, if you want to avoid this sort of problem (the designers of these packages did intend this restriction) then embed the generic part within an enclosing package, thus

```
package generic_IO is
 type File_Type is limited private;
 procedure Create(File : File_Type ....
 procedure Close .....
 generic
  Element_Type is private;
 package Read_Write is
  procedure Read(File : File_Type;
                 Element : Element_Type ...
  procedure Write .....
 end Read_Write;
end generic_IO;
```

Which would make our database package look something like

```
with generic_IO;
package A_Database is
 type File_Header is
  record
   Magic_Number      : Special_Stamp;
   Number_Of_Records : Record_Number;
   First_Deleted     : Record_Number;
  end record;
 type Row is
  record
   Key  : String(1 .. 80);
   Data : String(1 .. 255);
  end record;
 package Header_IO is new generic_IO.Read_Write (File_Header);
 use Header_IO;
 package Row_IO    is new generic_IO.Read_Write (Row);
 use Record_IO;
end A_Database;
:
:
declare
  Handle   : generic_IO.File_Type;
  A_Header : File_Header;
  A_Row    : Row;
begin
  generic_IO.Open(File => Handle, Name => "Test");
  Header_IO.Write(Handle, A_Header);
  Row_IO.Write(Handle, A_Row);
  generic_IO.Close(Handle);
end;
```

5.3 Streams

This is a new Ada-95 feature which I will add once I have a copy of GNAT which supports the feature. I like to have examples which I have compiled/tried.

6 Interfacing to other languages

Ada-95 has a specified set of packages under the top level package `Interfaces` which define functions to allow you to convert data types between the Ada program and the external language routines.

The full set of packages defined for interfaces are show below.

Interfaces

- C
  - Pointers
  - Strings
- COBOL
- CPP
- Fortran

7 Concurrency

To some this section does not fit in the remit of a C++ programmers guide to Ada, however most modern operating systems contain constructs known either as *lightweight processe*s or as *threads*. These allow programmers to have multiple threads of execution within the same address space. Many of you will be familiar with this concept and so I will use it as a basis for explaining tasks below, you may skip the next paragraph.

Unlike C/C++ Ada defines a concurrency model as part of the language itself. Some languages (Modula-3) provide a concurrency model through the use of standard library packages, and of course some operating systems provide libraries to provide concurrency. In Ada there are two base components, the task which encapsulates a concurrent process and the protected type which is a data structure which provides guarded access to its data.

7.1 Tasks

7.1.1 Tasks as threads

For those who have not worked in a multi-threaded environment you might like to consider the advantages. In a non-multi-threaded UNIX (for example) the granularity of concurrency is the process. This process is an atomic entity to communicate with other processes you must use sockets, IPC etc. The only way to start a cooperating process is to initialise some global data and use the `fork` function to start a process which is a copy of the current process and so inherits these global variables. The problem with this model is that the global variables are now replicated in both processes, a change to one is not reflected in the other.

In a multi-threaded environment multiple concurrent processes are allowed within the same address space, that is they can share global data. Usually there are a set of API calls such as `StartThread`, `StopThread` etc which manage these processes.

Note: An Ada program with no tasks is really an Ada process with a single running task, the default code.

7.1.2 A Simple task

In the example below an Ada task is presented which will act like a thread found in a multi-threaded operating system such as OS/2, Windows-NT or Solaris.

```
task X is
end X;

task body X is
begin
 loop
  -- processing.
 end loop;
end X;
```

As with packages a task comes in two blocks, the specification and the body. Both of these are shown above, the task specification simply declares the name of the task and nothing more. The body of the task shows that it is a loop processing something. In many cases a task is simply a straight through block of code which is executed in parallel, or it may be, as in this case, modelled as a service loop.

7.1.3 Task as types

Tasks can be defined as types, this means that you can define a task which can be used by any client. Once defined as a task objects of that type can be created in the usual way. Consider:

```
task type X is
end X;
Item : X;
Items : array (0 .. 9) of X;
```

Note: however that tasks are declared as constants, you cannot assign to them and you cannot test for equality.

7.2 Task synchronization (Rendezvouz)

The advantage of Ada tasking is that the Ada task model provides much more than the multi-threaded operating systems mentioned above. When creating a thread to do some work we must seperately create semaphores and/or other IPC objects to manage the cooperation between threads, and all of this is of course system dependant.

The Ada tasking model defines methods for inter-task cooperation and much more in a system independant way using constructs known as *Rendezvous*.

A Rendezvouz is just what it sounds like, a meeting place where two tasks arrange to meet up, if one task reaches it first then it waits for the other to arrive. And in fact a queue is formed for each rendezvous of all tasks waiting (in FIFO order).

7.2.1 entry/accept

A task contains a number of elements, data items, procedural code and rendezvous. A rendezvous is represented in the task specification like a procedure call returning no value (though it can have **`in out`** parameters). It can take any number of parameters, but rather that the keyword **`procedure`** the keyword **`entry`** is used. In the task body however the keyword **`accept`** is used, and instead of the procedure syntax of **`is begin`** simply **`do`** is used. The reason for this is that rendezvous in a task are simply sections of the code in it, they are not seperate elements as procedures are.

Consider the example below, a system of some sort has a cache of elements, it requests an element from the cache, if it is not in the cache then the cache itself reads an element from the master set. If this process of reading from the master fills the cache then it must be reordered. When the process finishes with the item it calls `PutBack` which updates the cache and if required updates the master.

```
task type Cached_Items is
 entry Request(Item : out Item_Type);
 entry PutBack(Item : in Item_Type);
end Cached_Items;

task body Cached_Items is
 Log_File : Ada.Text_IO.File_Type;
begin
 -- open the log file.
 loop
  accept Request(Item : out Item_Type) do
   -- satisfy from cache or get new.
  end Request;
  -- if had to get new, then quickly
  -- check cache for overflow.
  accept PutBack(Item : in Item_Type) do
   -- replace item in cache.
  end PutBack;
  -- if item put back has changed
  -- then possibly update original.
 end loop;
end Cached_Items;

-- the client code begins here:
declare
 Cache : Cached_Items;
 Item : Item_Type;
begin
 Cache.Request(Item);
 -- process.
 Cache.PutBack(Item);
end;
```

It is the sequence of processing which is important here, Firstly the client task (remember, even if the client is the main program it is still, logically, a task) creates the cache task which executes its body. The first thing the cache (owner task) does is some procedural code, its initialisation, in this case to open its log file. Next we have an **`accept`** statement, this is a rendezvous, and in this case the two parties are the owner task, when it reaches the keyword **`accept`** and the client task that calls `Cache.Request(Item)`.

If the client task calls `Request` before the owner task has reached the **`accept`** then the client task will wait for the owner task. However we would not expect the owner task to take very long to open a log file, so it is more likely that it will reach the **`accept`** first and wait for a client task.

When both client and owner tasks are at the rendezvous then the owner task executes the **`accept`** code while the client task waits. When the owner task reaches the end of the rendezvous both the owner and the client are set off again on their own way.

7.2.2 select

If we look closely at our example above you might notice that if the client task calls `Request` twice in a row then you have a deadly embrace, the owner task cannot get to `Request` before executing `PutBack` and the client task cannot execute `PutBack` until it has satisfied the second call to `Request`.

To get around this problem we use a **`select`** statement which allows the task to specify a number of entry points which are valid at any time.

```
task body Cached_Items is
 Log_File : Ada.Text_IO.File_Type;
begin
 -- open the log file.
 accept Request(Item : Item_Type) do
  -- satisfy from cache or get new.
 end Request;
 loop
  select
   accept PutBack(Item : Item_Type) do
    -- replace item in cache.
   end PutBack;
   -- if item put back has changed
   -- then possibly update original.
  or
   accept Request(Item : Item_Type) do
    -- satisfy from cache or get new.
   end Request;
   -- if had to get new, then quickly
   -- check cache for overflow.
  end select;
 end loop;
end Cached_Items;
```

We have done two major things, first we have added the **`select`** construct which says that during the loop a client may call either of the entry points. The second point is that we moved a copy of the entry point into the initialisation section of the task so that we must call `Request` before anything else. It is worth noting that we can have many entry points with the same name and they may be the same or may do something different but we only need one **`entry`** in the task specification.

In effect the addition of the **`select`** statement means that the owner task now waits on the **`select`** itself until one of the specified accepts are called.

Note: possibly more important is the fact that we have not changed the specification for the task at all yet!.

7.2.3 guarded entries

Within a select statement it is possible to specify the conditions under which an **accept** may be valid, so:

```
task body Cached_Items is
 Log_File : Ada.Text_IO.File_Type;
 Number_Requested : Integer := 0;
 Cache_Size : constant Integer := 50;
begin
 -- open the log file.
 accept Request(Item : Item_Type) do
  -- satisfy from cache or get new.
 end Request;
 loop
  select
   when Number_Requested > 0 =>
   accept PutBack(Item : Item_Type) do
     -- replace item in cache.
   end PutBack;
   -- if item put back has changed
   -- then possibly update original.
  or
   accept Request(Item : Item_Type) do
     -- satisfy from cache or get new.
   end Request;
   -- if had to get new, then quickly
   -- check cache for overflow.
  end select;
 end loop;
end Cached_Items;
```

This (possibly erroneous) example adds two internal values, one to keep track of the number of items in the cache, and the size of the cache. If no items have been read into the cache then you cannot logicaly put anything back.

7.2.4 delays

It is possible to put a **delay** statement into a task, this statement has two modes, delay for a given amount of time, or delay until a given time. So:

```
delay 5.0; -- delay for 5 seconds
delay Ada.Calendar.Clock; -- delay until it is ...
delay until A_Time; -- Ada-95 equivalent of above
```

The first line is simple, delay the task for a given number, or fraction of, seconds. This mode takes a parameter of type `Duration` specified in the package `System`. The next two both wait until a time is reached, the secodn line also takes a `Duration`, the third line takes a parameter of type `Time` from package `Ada.Calendar`.

It is more interesting to note the effect of one of these when used in a select statement. For example, if an **`accept`** is likely to take a long time you might use:

```
select
 accept An_Entry do
 end An_Entry;
or
 delay 5.0;
 Put("An_Entry: timeout");
end select;
```

This runs the **`delay`** and the **`accept`** concurrently and if the **`delay`** completes before the accept then the **`accept`** is aborted and the task continues at the statement after the **`delay`**, in this case the error message.

It is possible to protect procedural code in the same way, so we might amend our example by:

```
task body Cached_Items is
 Log_File : Ada.Text_IO.File_Type;
 Number_Requested : Integer := 0;
 Cache_Size : constant Integer := 50;
begin
 -- open the log file.
 accept Request(Item : Item_Type) do
   -- satisfy from cache or get new.
 end Request;
 loop
  select
   when Number_Requested > 0 =>
   accept PutBack(Item : Item_Type) do
     -- replace item in cache.
   end PutBack;
   select
    -- if item put back has changed
    -- then possibly update original.
   or
    delay 2.0;
    -- abort the cache update code
   end select;
  or
   accept Request(Item : Item_Type) do
     -- satisfy from cache or get new.
   end Request;
   -- if had to get new, then quickly
   -- check cache for overflow.
  end select;
 end loop;
end Cached_Items;
```

7.2.5 select else

The **else** clause allows us to execute a non-blocking **select** statement, so we could code a polling task, such as:

```
select
 accept Do_Something do
 end DO_Something;
else
 -- do something else.
end select;
```

So that if no one has called the entry points specified we continue rather than waiting for a client.

7.2.6 termination

The example we have been working on does not end, it simply loops forever. We can terminate a task by using the keyword **terminate** which executes a nice orderly cleanup of the task. (We can also kill a task in a more immediate way using the **abort** command, this is NOT recommended).

The **terminate** alternative is used for a task to specify that the run time environment can terminate the task if all its actions are complete and no clients are waiting.

```
loop
 select
  accept Do_Something do
  end Do_Something;
 or
  terminate;
 end select;
end loop;
```

The **abort** command is used by a client to terminate a task, possibly if it is not behaving correctly. The command takes a task identifer as an argument, so using our example above we might say:

```
if Task_In_Error(Cache) then
  abort Cache;
end if;
```

The **then abort** clause is very similar to the **delay** example above, the code between **then abort** and **end select** is aborted if the **delay** clause finishes first.

```
select
 delay 5.0;
 Put("An_Entry: timeout");
then abort
 accept An_Entry do
 end An_Entry;
end select;
```

7.2.7 conditional entry calls

In addition to direct calls to entry points clients may rendezvous with a task with three conditional forms of a select statement:

- Timed entry call
- Conditional entry call
- Asynchronous select

7.3 Protected types

Protected types are a new feature added to the Ada-95 language standard. These act like the monitor constructs found in other languages, which means that they monitor access to their internal data and ensure that no two tasks can access the object at the same time. In effect every entry point is mutually exclusive. Basically a protected type looks like:

```
protected type Cached_Items is
 function Request return Item_Type;
 procedure PutBack(Item : in Item_Type);
private
 Log_File : Ada.Text_IO.File_Type;
 Number_Requested : Integer := 0;
 Cache_Size : constant Integer := 50;
end Cached_Items;

protected body Cached_Items is
 function Request return Item_Type is
 begin
   -- initialise, if required
   -- satisfy from cache or get new.
   -- if had to get new, then quickly
   -- check cache for overflow.
 end Request;
 procedure PutBack(Item : in Item_Type) is
 begin
   -- initialise, if required
   -- replace item in cache.
   -- if item put back has changed
   -- then possibly update original.
 end Request;
end Cached_Items;
```

This is an implementation of our cache from the task discussion above. Note now that the names `Request` and `PutBack` are now simply calls like any other. This does show some of the differences between tasks and protected types, for example the protected type above, because it is a passive object cannot completly initialise itself, so each procedure and/or function must check if it has been initialised. Also we must do all processing within the stated procedures.

References

[1] Ada Language Reference Manual `http://www.adahome/rm95`

[2] Ada Rationale `http://www.adahome.com/LRM/95/Rationale/rat95html/rat95-contents.html`

[3] Ada Quality and Style: Guidelines for professional programmers

[4] Programming in Ada (3rd Edition), J.G.P.Barnes, Addison Wesley.

[5] Ada Programmers FAQ `http://www.adahome.com/FAQ/programming.html#title`.

[6] Abstract Data Types Are Under Full Control with Ada9X (TRI-Ada ’94) `http://www.adahome.com/Resources/Papers.html`

[7] Working With Ada9X Classes (TRI-Ada ’94) `http://www.adahome.com/Resources/Papers.html`

[8] Lovelace on-line tutorial `http://www.adahome.com/Tutorials/Lovelace/lovelace.htm`.

[9] Design and evolution of C++, Bjarne Stroustrup, Addison Wesley.

[10] The annotated C++ reference manual, Margaret Ellis and Bjarne Stroustrup, Addison Wesley.

Acknowledgements

My thanks must go to the following people for help, pointers, proof-reading, suggestions and encouragement.

S. Tucker Taft (Intermetrics), Magnus Kempe and Robb Nebbe (EPFL), Michael Bartz (University of Memphis), David Weller, Kevin Nash (Bournemouth University), Laurent Guerby, Jono Powell, Bill Wagner (Hazeltine).

Once again, thank you.

For comments, additions, corrections, gripes, kudos, etc. e-mail to:

Simon Johnston (Team Ada) – `skj@rb.icl.co.uk`

ICL Retail Systems

Index