Modern Programming Languages

Lecture 13-17

Ada Programming Language
An Introduction

Design Goals

Ada is a computer programming language originally designed to support the construction of long-lived, highly reliable software systems. Its design emphasizes readability, avoids error-prone notation, encourages reuse and team coordination, and it is designed to be efficiently implementable.
A significant advantage of Ada is its reduction of debugging time. Ada tries to catch as many errors as reasonably possible, as early as possible. Many errors are caught at compile-time by Ada that aren't caught or are caught much later by other computer languages.

Ada programs also catch many errors at run-time if they can't be caught at compile-time (this checking can be turned off to improve performance if desired).

In addition, Ada includes a problem (exception) handling mechanism so that these problems can be dealt with at run-time.

The main design goals of Ada were:

- Program reliability and maintenance
- Military software systems are expected to have a minimum lifetime of 30 years.
- Programming as a human activity
- Efficiency

Hence emphasis was placed on program readability over ease of writing.

Ada History

The need for a single standard language was felt in 1975 and the draft requirements were given the code name strawman. Strawman was refined to Woodman and then Tinman in 1976. It was further refined to ironman. At that time proposals were invited for the design of a new language. Out of the 17 proposals received, four were selected and given the code names of green, red, blue, and yellow. Initial designs were submitted in 1978 and red and green short listed on the basis of these designs. Standard requirements were then refined to steelman. The designs were refined further and finally Green was selected in 1979. DoD announced that the language will be called Ada. The 1995 revision of Ada (Ada 95) was developed by a small team led by Tucker Taft. In both cases, the design underwent a public comment period where the designers responded to public comments.

Ada Features

The salient features of Ada language are as follows:

- Packages (modules) of related types, objects, and operations can be defined.
- Packages and types can be made generic (parameterized through a template) to help create reusable components.
- It is strongly typed
- Errors can be signaled as exceptions and handled explicitly. Many serious errors (such as computational overflow and invalid array indexes) are automatically caught and handled through this exception mechanism, improving program reliability.
- Tasks (multiple parallel threads of control) can be created and communicate. This is a major capability not supported in a standard way by many other languages.
- Data representation can be precisely controlled to support systems programming.
- A predefined library is included; it provides input/output (I/O), string manipulation, numeric functions, a command line interface, and a random number generator (the last two were available in Ada 83, but are standardized in Ada 95).
- Object-oriented programming is supported (this is a new feature of Ada 95). In fact, Ada 95 is the first internationally standardized object-oriented programming language.
- Interfaces to other languages (such as C, Fortran, and COBOL) are included in the language.

The first Example – Ada "Hello World"

```
with Ada.Text_Io;          -- intent to use
use Ada.Text_Io;           -- direct visibility

                           -- the first two statements are kind of include in C

procedure Hello is         -- procedure without parameters is the
                           -- starting point
begin
        Put_Line("Hello World!");
                           -- this statement prints "Hello World" on the output
end Hello;
```

It may be noted that Ada is not case sensitive.

Ada Operators

Ada has a rich set of operators. The following table gives a list of these operators and also shown corresponding C++ operators for reference.

Operator	C/C++	Ada
Assignment	=	:=
Equality	==	=
Non Equality	!=	/=
Greater Than	>	>
Less Than	<	<
Greater Than Or Equal	>=	>=
Less Than Or Equal	<=	<=
PlusEquals	+=	
SubtractEquals	-=	
MultiplyEquals	*=	
DivisionEquals	/=	
OrEquals	|=	
AndEquals	&=	
Modulus	%	Mod
Remainder		Rem
AbsoluteValue		Abs
Exponentiation		**
Range		..
Membership		In
Logical And	&&	And
Logical Or	||	Or
Logical Not	!	Not
Bitwise And	&	And
Bitwise Or	|	Or
Bitwise Exclusive Or	^	Xor
Bitwise Not	~	Not
String Concatenation		&

It is important to note that Ada has not included operators like PlusEquals as such operators reduce readability.

Operator Overloading

Ada allows a limited overloading of operators. The exception in Ada is that the assignment operator (**:=**) cannot be overridden. It can be overridden in case of inheritance from a special kind of “abstract class”. When you override the equality operator (**=**) you also implicitly override the inequality operator (***/=***).

Ada Types

Ada provides a large number of kinds of data types. Ada does not have a predefined inheritance hierarchy like many object oriented programming languages. Ada allows you to define your own data types, including numeric data types. Defining your own type in Ada creates a new type.

Elementary Types

The elementary Ada type are:

- Scalar Types
- Discrete Types
- Real Types
- Fixed Point Types
- Access Types

Discrete Types

Discrete types include Integer types, Modular types, Character types, enumeration types, and Boolean type.

Integer Types

We first look at Signed Integer types. Ada, like other languages, supports signed integers. However, in this languages we can also define our own integer type with a limited set of values as shown in the following example:

type Marks **is range** 0..100;

This defines a type Marks with the property that a variable of this type can only have values between 0 and 100.

We can now create variable with this type as shown below:

finalScore : Marks;

You may also note that the syntax of Ada for variable declaration is different from C. In this case, the type comes after the variable name and is separated by a : from the variable names.

Unsigned (Modular) Types

Modular types support modular arithmetic and have wrap around property. This concept is elaborated with the help of the following example:

```
type M is mod 7; -- values are 0,1,2,3,4,5,6

q : m := 6; -- initialization
…
q := q + 2; -- result is 1
```

It is most commonly used as conventional unsigned numbers where overflows and underflows are wrapped around.

```
type Uns_32 is mod 2 ** 32;
```

Remember that twos complement arithmetic is equivalent to mod 2**wordsize.

Character Types

There are two built in character types in Ada

- Simple 8-bit ASCII Characters
- Wide_Character that support 16-bit Unicode/ISO standard 10646.

These are good enough for most purposes, but you can define your own types just like the integer types:

```
type LetterGrades is ('A', 'B', 'C', 'D', 'F', 'I', 'W');
```

This can now be used to declare variables of this type.

Enumeration Types

Just like C, an enumeration type in Ada is a sequence of ordered enumeration literals:

```
type Colors is (Red, Orange, Yellow, Green, Blue, Indigo, Violet);
type State is (Off, Powering_Up, On);
```

It is however different from C in many respects:

1. There is no arithmetic defined for these types. For example:
   ```
   S1, S2 : State;
   S1 := S1 + S2; -- Illegal
   ```

2. One can however add/subtract one (sort of increment and decrement) using the Pred and Succ as shown below:

   State'Pred (S1)
   State'Succ (S2)

3. Unlike C, the same symbolic literal can be used in two enumeration types. For example:

   type RainbowColors **is** (Red, Orange, Yellow, Green, Blue, Indigo, Violet);
   type BasicColors **is** (Red, Green, Blue);

The Ada compiler will use the type of the variable in question and there will be no ambiguity.

Enumeration types are used as attributes or properties of different objects.

Boolean Types

Boolean is a predefined enumeration type and has the following definition.

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

Expressions of type Boolean are used in logical statements and are used as conditions in the **if** statements, **while** loops, **exit** statements, etc.

Floating Point Types

Like most other languages, Ada also supports floating point types. In this case, we can also explicitly declare the desired precision of the number. For example:

type Double **is digits** 15;

defines a floating point type with 15 decimal digits of precision.

type Sin_Values **is digits** 10 **range** -1.0..1.0;

defines a type with 10 decimal digits of precision and a valid range of values from -1.0 through 1.0.

Ordinary Fixed Point Types

Ada also supports fixed point real numbers that are used for more precise decimal arithmetic such as financial applications. In the Ordinary fixed point type, the distance between values is implemented as a power of 2. For example:

type Batting_Averages **is delta** 0.001 **range** 0.0..1.0;

The type *Batting_Averages* is a fixed point type whose values are evenly spaced real numbers in the range from 0 through 1. The distance between the values is no more than 1/1000. The actual distance between values may be 1/1024, which is 2^{-10}.

Decimal Fixed Point Types

In the case of decimal fixed point types, the distance between values is implemented as a power of 10.

type Balances **is delta** 0.01 **digits** 9 **range** 0.0 .. 9_999_999.99;

The important difference in the definition of a decimal fixed point type from an ordinary fixed point type is the specification of the number of digits of precision.

This example also shows how Ada allows you to specify numeric literals with underscores separating groups of digits. The underscore is used to improve readability.

Arrays

Like most other languages, Ada also supports arrays. One major difference between the arrays in Ada and C is that the indexes of arrays in Ada do not start from 0. In fact, the range of indexes to be used can be defined by the programmer as shown below:

type Int_Buffer **is array** (1..10) **of** Integer;

type Normalized_Distribution **is array**(-100..100) **of** Natural;

An array type can also be defined without a specific size. In this case the size is defined at the time of instantiation of a variable of that type. For example:

type String **is array** (Positive **range** <>) **of** Character;

Can now be used to create variable of specific size as shown below:

Last_Name : String(1..20);

Another very interesting feature of Ada is that the array indexes can be of any discrete type. For example if we define Days as below:

type Days **is** (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday);

then we can create an array which uses Days (or any sub-range of Days) as the array indexes as shown below.

type Daily_Sales **is array**(Days) **of** Float;

Now the type Daily_Sales is an array type of size 7 and indexes Monday through Sunday. Also note that these literals will be used as indexes in the statements that reference array elements.

Record Definitions

Record types are like structures in C. The following example shows a record type Address with three fields.

```
type Address is record
        Street : Unbounded_String;
        City :    Unbounded_String;
        Postal_Code : String(1..10);
end record;
```

Once defined, it can be used to declare variables of that type as shown below:

My_Address : Address;

The dot operator, '.', is used to access individual fields. This is demonstrated in the example below:

My_Address.Postal_Code := "00000-0000";

Discriminated Records

Discriminated records are like union types in C. There are however major differences between C union types and Ada discriminated records. The union type in C is fundamentally unsafe, and therefore unacceptable. For example, consider the following definition:

union (**int**, **float**) puzzle;

Now puzzle has either an **int** or a **float.** But at runtime, the compiler and the environment cannot tell which particular value has been stored currently. So we have to trust the programmer. In Ada we are short on trust. The whole philosophy is that a programmer is a human and all humans make mistakes. So to make it safer, Ada uses discriminants in a record that tell what type of data is currently stored there. This is shown in the following example:

```
type IF is (Int, Float);
type Int_Float (V : IF := Int) is record
  case V is
    when Int => Int_Val : Integer;
    when Float => Float_Val : Float;
  end case;
end record;
```

Now the value of the discriminant V shows what type is currently present. Variables of a discriminated type are referenced as shown in the following example:

```
Puzzle : Int_Float;
…
Puzzle := (Float, 1.0);
F := Puzzle.Float_Val;          -- OK
I := Puzzle.Int_Val;            -- type mismatch - raise exception
…
Puzzle.V := SomeIntValue;       -- not allowed!
```

Discriminated records can be used for making subtypes

```
subtype PuzzleI is puzzle (Int);          -- this type only holds int values
```

These can also be used to specify array dimensions:

```
Subtype Vlen is Integer range 1 .. 10;
type Vstr (Vlen : Integer := 0) is record
  Data : String (1 .. Vlen);
end record;
VV : Vstr := (5, “hello”);
```

Access Types

An *access* type roughly corresponds to a C++ pointer.

type Address_Ref **is access** Address;

```
A_Ref := new Address;
A_Ref.Postal_Code := "94960-1234";
```

Note that, unlike C, there is no notational difference for accessing a record field directly or through an access value.

To refer to the entire record accessed by an access value use the following notation:

```
Print(A_Ref.all);
```

Statement Forms

Assignment statement

Like all imperative languages, Ada also supports the assignment statement. However, in Ada the assignment is *not* an expression like C. The syntax of the assignment statement is as follows:

Variable := *expression*;

Note that, in Ada, ':=' is used as the assignment operator whereas '=' is used as assignment operator in C.

If Statement

The Ada if statement is fully blocked. In fact, all Ada control structures are fully blocked. That means, we do not have two separate forms if we need only one statement to be executed if the condition for the if statement is true and if we want more than one statements. The body of the if statements is always in side a a complete block that starts with the **then** keyword and end with **end if** as shown below:

```
if condition then
        -- statement(s)
end if;
```

If an if statement has multiple else parts with separate conditions, then Ada uses the keyword **elsif** to indicate that. The **else** part is optional and comes at the end. Each **if** statement must have a corresponding **end if** in it. The following examples demonstrate this concept.

```
if condition1 then
    -- statement(s)
elsif condition2 then
    if condition3 then
        -- statements
    end if;
    -- statement(s)
…
else
    -- statement(s)
end if;
```

```
if condition1 then
    -- statements
elsif condition2 then
    if condition3 then
        -- statements
    end if;
    -- statements
…
else if condition4 then
    -- this is a new if statement and hence must have its own end if
    -- statements
  end if;
    -- statements
end if;
```

Case Statements

Case statement is like the switch statement in C, but as compared to C, it is safer and more structured. Its syntax is as shown below:

case *expression* **is**
    when *choice list* =>
        sequence-of-statements
    when *choice list* =>
        sequence-of-statements
    when others =>
        sequence-of-statements
end case;

As opposed to C, the case statement is fully structured; there is no equivalent of the break statement and the control does not fall through to the next choice after executing the set of statements for the selected choice. The choice _list can have more than one values specified in the form of a range specified with the .. operator like 1..10, discrete values separated by | such as a | e | i | o | u, or a combination of both. The following example elaborates this concept.

```
case TODAY is
    when MON .. THU =>
        WORK;
    when FRI =>
        WORK;       PARTY;
    when SAT | SUN =>
        REST;
end case;
```

All values in **when** branches must be static and all possible values of expression must be included exactly once. In Ada, the **case** expression must match one of the choices given in the **when** clause as an exception will be raised as run time if there is no match. **when others** => is like **default** in C and is used to cover all the rest of the choices not mentioned in the above when clauses.

Looping

There are three types of loops in Ada:
1. Simple Loop – unconditional infinite loop
2. While loop
3. For Loop

Simple Loop

A simple loop is used to signify a potentially infinite loop. These loops are typically used in operating systems and embedded systems. Its syntax is as follows:

```
loop
        -- Loop body goes here
end loop;
```

The **exit** statement can be used to come out of the loop as shown below.

```
loop
        -- Loop body goes here
        exit when condition;
end loop;
```

It is equivalent to the following code segment.

```
loop
        -- Loop body goes here
        if condition then
                exit;
        end if;
end loop;
```

The **exit** statement is like the **break** statement in C but there are certain differences and will be discussed later.

While and For Loops

Ada has only the pre-test while loop and does not support the post-test loop like the **do-while** statement in C. The **while** statement in Ada has the following structure.

while *condition* **loop**
    -- Loop body goes here
end loop;

The semantics of the **while** statement are exactly the same as its counterpart in C.

The **for** statement in Ada is however quite different here. In C, the **for** statement is virtually the same as the **while** statement as the condition in the C **for** statement is checked in each iteration and the number of iterations therefore cannot be determined upfront.

In Ada **for** statement, the number of iterations are fixed the first time the control hits the **for** statement and is specified by a range just like the choice_list in the **case** statement as shown below:

for variable **in** low_value .. high_value **loop**
    -- Loop body goes here
end loop;

The value of the loop variable can be used but cannot be changed inside the loop body. The loop counting can be done in the reverse order as well. This is shown below.

for variable **in reverse** high_value .. low_value **loop**
    -- Loop body goes here
end loop;

Exit statement

The **exit** statement can be used with our without a **when** condition as mentioned in the case of the loop statement.
exit;
exit when condition;

It can only be used in a loop. It can use labels and can be used to specify the specific loop to exit as shown below.

```
Outer : loop
  Inner : loop
        …
        exit Inner when Done;
  end loop Inner;
end loop Outer;
```

Block Statement

Block statement in Ada is very similar to a block in C. It can be used anywhere and has the following structure:

```
declare          -- declare section optional
  declarations
begin
  statements
exception        -- exception section optional
  handlers
end;
```

Return statement

The **return** statement can only be used in subprograms and has the following form:

```
return;                    -- procedure
return expression;         -- function
```

Function must have at least one **return.**

Raise statement

Raise statement is like throw statement in C++ and is used to raise exception as shown below:

raise *exception-name*;

Declaring and Handling Exceptions

Ada was the first language to provide a structured exception handling mechanism. C++ exception handling is very similar to Ada. In Ada, the programmer can declare, raise, and handle exception.

Declaring an exception

Exceptions are declared just like any other variable before they can be raised. This is shown in the following example:

Error, Disaster : **exception**;

Raising an exception

Like **throw** in C++, an Ada programmer can explicitly raise an exception and it strips stack frames till a handler is found.

raise Disaster;

Handling an exception

Like C++, an exception handling code may be written at the end of a block.

Anywhere we have **begin-end**, we can also have an exception handling code. The structure of exception handling part is shown in the following example.

```
begin
  statements
exception
  when exception1 => statements;
        when exception2 => statements;
end;
```

Subprograms

In Ada, there are two types of subprograms: procedures and functions. Unlike C, we can also have nesting of subprograms. A subprogram defined inside another subprogram an be used by its parent, children, or sibling subprograms only.

This is demonstrated in the following example:

```
Procedure My_procedure(x, y : IN OUT integer) is
        …
        function aux_func(a : Integer) return integer is
                Temp : float;
        begin
                …
                return integer(temp) * a;  -- retrun temp * a is not allowed
        end aux_func;
        …
begin                   -- begin of My_procedure
        …
        aux_func(x);
        …
end My_procedure;
```

Packages

The primary structure used for encapsulation in Ada is called a *package*. Packages are used to group data and subprograms. Packages can be hierarchical, allowing a structured and extensible relationship for data and code. All the standard Ada libraries are defined within packages.

Package definitions usually consist of two parts: package specification and package body. Package bodies can also declare and define data types, variables, constants, functions, and procedures not declared in the package specification. In the following example, a STACK package is defined. We first the have the package specification as follows:

```
package STACK is
        procedure PUSH (x : INTEGER);
        function POP return INTEGER;
end STACK;
```

The body of the package is shown below.

```
package body STACK is
        MAX: constant := 100;
        S : array (1..MAX) of  INTEGER;
        TOP : INTEGER range 0..MAX;

        procedure PUSH (x : INTEGER) is
        begin
                TOP := TOP + 1;
                S(TOP) := x;
        end PUSH;

        function POP return integer is
                TOP := TOP - 1;
                return S(TOP + 1);
        end POP;
begin
        TOP := 0;
end STACK;
```

Once defined, a package can be used in the client program as follows:

```
with STACK;        use STACK;
procedure myStackExample is
        I, O : integer;
begin
        …
        push(i);
        …
        O := pop;
        …
end myStackExample;
```

Object-Orientation – The Ada Way

Ada provides tools and constructs for extending types through inheritance. In many object oriented languages the concept of a class is overloaded. It is both the unit of encapsulation and a type definition. Ada separates these two concepts. Packages are used for encapsulation and Tagged types are used to define extensible types.

Tagged Type

A tagged type is like a record which can be used to declare objects. Following is an example of a tagged type:

```
type Person is tagged record
        Name : String(1..20);
        Age : Natural;
end record;
```

Such a type definition is performed in a package. Immediately following the type definition must be the procedures and functions comprising the *primitive operations* defined for this type. *Primitive operations* must have one of its parameters be of the tagged type, or for functions the return value can be an instance of the tagged type. It allows you to overload functions based upon their return type. It may be noted that C++ and Java do not allow you to overload based upon the return type of a function.

Extending Tagged Types

Just like classes in C++, the tagged type can be extended by making a *new* tagged record based upon the original type as shown below:

```
type Employee is new Person with record
        Employee_Id : Positive;
        Salary : Float;
end record;
```

In this example, type *Employee* inherits all the fields and primitive operations of type *Person*.

Generics

Generics are like templates in C++ and allow parameterization of subprograms and packages with parameters which can be types and subprograms as well as values and objects. The following example elaborates the concept of generics.

```
generic
        MAX : positive;
        type item is private;
package STACK is
        procedure PUSH (x : item);
        function POP return item;
end STACK;
```

Note that the type of the item is left unspecified. The corresponding body of STACK package is as follows:

```
package body STACK is
        S : array (1..MAX) of  item;
        TOP : INTEGER range 0..MAX;

        procedure PUSH (x : item) is
        begin
        …
        end PUSH;

        function POP return item is
        …
        end POP;
begin
        TOP := 0;
end STACK;
```

Once we have the generic definition, we can instantiate our own STACK for the given type which is **integer** in the following example.

```
declare
        package myStack is new STACK(100, integer);
        use myStack;
begin
        …
        push (i);
        …
        O := pop;
        …
end;
```

Concurrency

Ada Tasking allows the initiation of concurrent tasks which initiates several parallel activities which cooperate as needed. Its syntax is similar to packages as shown below:

```
task thisTask is
        …-- interfaces
end thisTask;

task body thisTask is
        …
end thisTask;

task simpleTask;   -- this task does not provide an interface to other tasks
```

Here is a very simple example that demonstrates the concept of parallel processing. In this example we specify COOKING a procedure composed of three steps.

```
procedure COOKING is
begin
        cookMeat;
        cookRice;
        cookPudding;
end COOKING;
```

As these steps can be carried out in parallel, we define tasks for these tasks as shown below:

```
procedure COOKING is
        task cookMeat;
        task body cookMeat is
        begin
                -- follow instructions for cooking meat
        end cookMeat;

        task cookRice;
        task body cookRice is
        begin
                -- follow instructions for cooking rice
        end cookRice;

begin
        cookPudding;
end COOKING;
```

Other Features

Ada is a HUGE language and we have looked at may be only 25-30% of this language. It is to be remembered that the main objective of the Ada design was to incorporate the contemporary software engineering knowledge in it. In general, software development time is: 10% design, 10% coding, 60% debug and 20% test. Last 80% of the project is spent trying to find and eliminate mistakes made in the first 20% of the project. Therefore Ada promised to spend 20% time in design, 60% in coding, 10% in debug, and 10% in testing. Therefore cutting the entire cycle time in half!

I would like to finish with a quote from Robert Dewar:

> *When Roman engineers built a bridge, they had to stand under it while the first legion marched across. If programmers today worked under similar ground rules, they might well find themselves getting much more interested in Ada.*