



university.adacore.com



Range Checks and Overflows

Ranges

 Ada types are associated with a range which can be smaller than the underlying representation

```
type T1 is range 1 .. 10;
```

 The above means that all values of T1 stored have to be within the specified range

 Ranges are only checked on certain points in the program, like assignments. They are not checked on sub expressions.

```
V: T1 := 10;

V2 : T1 := 1 + V - 1; -- OK

V3 : T1 := 1 + V; -- EXCEPTION
```

Where are range checks performed?

On an assignment / explicit initialization

```
V : T1 (:=)2;
```

• On a conversion / qualification

```
V : Integer := Integer (1 + T1 (2));
```

• On a parameter passing

```
procedure P (V : in out T1);
```

Overflows

 Temporary results may be computed in a representation allowing for bigger temporary values than the type itself

 If the temporary value goes beyond the representation, then an overflow will occur

 If overflow checks are inactive and the resulting value is back in range, the result will be erroneous

Example of Failed Overflow Check

• ... with a little bit of complexity to fool the compiler ...

```
with Ada. Text IO; use Ada. Text IO;
procedure Main is
   V : Integer := Integer'Last;
   V2 : Integer := 10;
  procedure P is
  begin
     V := (V + 10) / 2;
      Put Line (Integer'Image (V));
      -- print -1073741819
   end P;
begin
 P;
end Main;
```

There are ways to avoid the above covered in a later lesson

Subtypes

A type is a consistent semantic entity

 A subtype is a special designation of this type, that may be associated with additional constraints but is not a new type

 Operations between a type and its subtypes are allowed, possibly with additional checks

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

V1 : Integer := 0;
V2 : Natural := V1; -- OK
V3 : Positive := V1; -- BAD, exception
```

Example of Subtype Usage: Naming a Constraint

```
type Day is (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday);
subtype Business is Day range Monday .. Friday;
subtype Weekend is Day range Saturday .. Sunday;
procedure Process Day (D : Day) is
begin
   if D in Business then
      Put Line ("Wake up, 7:00");
   elsif D in Weekend then
      case Weekend'(D) is
         when Saturday =>
            Put Line ("No Time Constraints");
         when Sunday =>
             Put Line ("Go to bed, 9:00 PM");
       end case;
   end if;
end Process Day;
```



Primitives

The notion of a primitive

- A type is characterized by two sets of properties
 - Its data structure
 - The set of operations that applies to it
- These operations are called methods in C++, or Primitives in Ada

```
type T is record
    Attribute_Data : Integer;
end record;

procedure Attribute_Function (This : T);

Ada

C++

class T {
    public:
        int Attribute_Data;
        void Attribute_Function (void);
};
```

In Ada

- the primitive relationship is implicit
- The "hidden" parameter "this" is explicit
 (and can have any name)

General rule for a primitive

A subprogram S is a primitive of type T if

- S is declared in the scope of T
- S has at least one parameter of type T (of any mode, including access) or return a value of type T

```
package P is

type T is range 1 .. 10;

procedure P1 (V : T);
procedure P2 (V1 : Integer; V2 : T);
function F return T;
end P;
```

A subprogram can be a primitive of several types

```
package P is

  type T1 is range 1 .. 10;
  type T2 is (A, B, C);

procedure Proc (V1 : T1; V2 : T2);
end P;
```

Implicit primitive operations

 At type declaration, primitives are implicitly created if not explicitly given by the developer, depending on the kind of the type

```
type T1 is range 1 .. 10;
-- implicitly declares function "+" (Left, Right : T1) return T1;
-- implicitly declares function "-" (Left, Right : T1) return T1;
-- ...

type T2 is null record;
-- implicitly declares function "=" (Left, Right : T2) return T2;
end P;
```

These primitives can be used just as any others

```
procedure Main is
    V1, V2 : P.T1;
begin
    V1 := P."+" (V1, V2);
end Main;
```

The "use all type" clause

 Often, "use clauses" are forbidden by a coding standard. This means that all operations have to be prefixed

```
package Parent.Child.A is
    type T1 is range 1 .. 10;
    procedure Print (V : T1);
end Parent.Child.A;

with Parent.Child.A;

procedure Main is
    V1 : Parent.Child.A.T1 := 2;
    V2 : Parent.Child.A.T1 := 2;
begin
    V1 := Parent.Child.A."+" (V1, V2);
    Parent.Child.A.Print (V1);
end Main;
```

 Many coding standards allow "use type clauses" which give visibility only on the primitives of given types

```
with Parent.Child.A; use all type Parent.Child.A.T1;

procedure Main is
    V1 : Parent.Child.A.T1 := 2;
    V2 : Parent.Child.A.T1 := 2;

begin
    V1 := V1 + V2; -- allowed by use type
    Print (V1); -- allowed by use all type
end Main;
```



Derived Types

Simple type derivation

In Ada, any (non-tagged) type can be derived

```
type Child is new Parent;
```

- A child is a distinct type inheriting from:
 - The data representation of the parent
 - The primitives of the parent

```
type Parent is range 1 .. 10;
procedure Prim (V : Parent);

type Child is new Parent;
-- implicit procedure Prim (V : Child);

V : Child;
begin
V := 5;
Prim (V);
```

Conversions are possible for non-primitive operations

```
package P is
   type Parent is range 1 .. 10;
   type Child is new Parent;
end P;
```

```
procedure Main is
    procedure Not_A_Primitive (V : Parent);

V1 : Parent;
    V2 : Child;

begin
    Not_A_Primitive (V1);
    Not_A_Primitive (Parent (V2));
end Main;
```

What can simple derivation do to the structure?

- The structure of the type has to be preserved
 - An array stays an array
 - A scalar stays a scalar
- Scalar ranges can be reduced

```
type Int is range -100 .. 100;
type Nat is new Int range 0 .. 100;
type Pos is new Nat range 1 .. 100;
```

Constraints on unconstrained types can be specified

```
type My_Array is array (Integer range <>) of Integer;

type Ten_Elem_Array is new My_Array (1 .. 10);

type Rec (Size : Integer) is record
    Elem : My_Array (1 .. Size);
end record;

type Rec_With_Ten_Elem_Array is new Rec (10);
```

What can simple derivation do to the list of operations?

 Operations can be overridden – this overriding can be checked by the optional "overriding" reserved word

```
type Root is range 1 .. 100;
procedure Prim (V : Root);

type Child is new Root;
overriding procedure Prim (V : Child);
```

Operations can be added – this addition can be checked by the optional "not overriding" reserved word

```
type Root is range 1 .. 100;
procedure Prim (V : Root);

type Child is new Root;
not overriding procedure Prim2 (V : Child);
```

 Operations can be removed – the removal can be checked by the optional "overriding" reserved word

```
type Root is range 1 .. 100;
procedure Prim (V : Root);

type Child is new Root;
overriding procedure Prim (V : Child) is abstract;
```





Is this correct? (1/10)



```
type T1 is range 1 .. 10;

V : T1 := 10;
begin
V := (1 + V) - 1;
```

Is this correct? (1/10)



```
type T1 is range 1 .. 10;

V : T1 := 10;
begin
V := (1 + V) - 1;
```

1 + V = 11, but this is a temporary result 11 - 1 = 10, then assigned to V No problem

Is this correct? (2/10)



```
type T1 is range 1 .. 10;

V : T1 := 10;
begin
V := T1'(1 + V) - 1;
```

Is this correct? (2/10)



```
type T1 is range 1 .. 10;

V : T1 := 10;
begin
V : T1'(1 + V) - 1;
```

The qualification to T1 will verify T1 constraints as 1 + 10 = 11, outside of the constraints => Constraint_Error

Is this correct? (3/10)



```
V1 : Integer := Integer'Last;
V2 : Integer := Integer'Last;
begin
V1 := (V1 * 2) / 4;
V2 := V2 * (2 / 4);
```

Is this correct? (3/10)



```
V1 : Integer := Integer'Last;
  V2 : Integer := Integer'Last;
begin
   V1 := (V1 * 2)
   V2 := V2
```

This first statement has an overflow. The second is fine though, due to operator priority.

Is this correct? (4/10)



```
type T is new Integer range 0 .. Integer'Last;
V1 : Integer := 0;
V2 : T := V1;
```

Is this correct? (4/10)



```
type T is new Integer range 0 .. Integer'Last;

V1 : Integer := 0;
V2 : T := V1;
```

Type consistency error. Integer and T are two different types

Is this correct? (5/10)



```
subtype T is Integer range 1 .. Integer'Last;
V1 : Integer := 0;
V2 : T := V1;
```

Is this correct? (5/10)



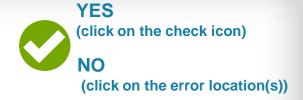
```
subtype T is Integer range 1 .. Integer'Last;

V1 : Integer := 0;
V2 : T := V1;
```

T is a subtype of Integer. Therefore, V1 and V2 are of the same type.

Is this correct?

(6/10)



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

Is this correct? (6/10)

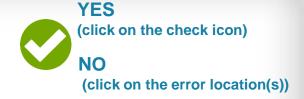


```
subtype Natural is Positive range 0 .. Positive Last;
```

A subtype can't extend the parent range, only reduce it.



Is this correct? (7/10)



procedure Proc (V : T1);

with P1; use P1;

package P3 is

end P3;

```
package P1 is
   type T1 is range 1 .. 10;
end P1;
```

```
with P1; use P1;
package P2 is
   type T2 is new T1;
end P2;
```

```
with P1; use P1;
with P2; use P2;
with P3; use P3;

procedure Main is
   V : T2;
begin
   Proc (V);
end Main;
```

Is this correct? (7/10)



```
package P1 is
    type T1 is range 1 .. 10;
end P1;
```

```
with P1; use P1;
package P2 is
   type T2 is new T1;
end P2;
```

```
with P1; use P1;
package P3 is
    procedure Proc (V : T1);
end P3;
```

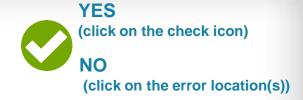
```
with P1; use P1;
with P2; use P2;
with P3; use P3;

procedure Main is
   V : T2;
begin
   Proc (V);
end Main;
```

T1 and T2 are two different types Proc only applies to T1

write:
Proc (T1 (V));
instead

Is this correct? (8/10)



```
package P1 is
   type T1 is range 1 .. 10;
  procedure Proc (V : T1);
end P1;
```

```
with P1; use P1;
package P2 is
   type T2 is new T1;
end P2;
```

```
with P1; use P1;
with P2; use P2;
procedure Main is
  V : T2;
begin
   Proc (V);
end Main;
```

Is this correct? (8/10)



```
package P1 is
   type T1 is range 1 .. 10;
   procedure Proc (V : T1);
end P1;
```

```
with P1; use P1;
package P2 is
    type T2 is new T1;
end P2;
```

```
with P1; use P1;
with P2; use P2;

procedure Main is
   V : T2;
begin
   Proc (V);
end Main;
```

In this case, Proc is a inherited primitive of T2, so it can be directly called.



What's the output of this code? (9/10)

```
type T1 is range 1 .. 10;
procedure Proc (V : T1);

type T2 is range 1 .. 10;
procedure Proc (V : T2);
end P;
```

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

package body P is

procedure Proc (V : T1) is
begin
    Put ("1 ");
end Proc;

procedure Proc (V : T2) is
begin
    Put ("2 ");
end Proc;

end Proc;
```

```
with P; use P;

procedure Main is
    V1 : T1;
    V2 : T2;

begin
    Proc (V1);
    Proc (V2);
    Proc (T2 (V1));
    Proc (T1 (V2));
end Main;
```



What's the output of this code? (9/10)

```
type T1 is range 1 .. 10;
procedure Proc (V : T1);

type T2 is range 1 .. 10;
procedure Proc (V : T2);
end P;
```

with Ada.Text_IO; use Ada.Text_IO;

package body P is

procedure Proc (V : T1) is
begin
 Put ("1 ");
end Proc;

procedure Proc (V : T2) is
begin
 Put ("2 ");
end Proc;

end Proc;

In case of a conversion, the target of the conversion is the type of the expression. So the result is:

1221

```
with P; use P;

procedure Main is
    V1 : T1;
    V2 : T2;

begin
    Proc (V1);
    Proc (V2);
    Proc (T2 (V1));
    Proc (T1 (V2));
end Main;
```

Is this correct? (10/10)



```
package P is
   type T1 is range 1 \dots 10;
   procedure Proc (V : T1);
   type T2 is new T1;
   type T3 is new T2;
   overriding procedure Proc (V : T3);
end P;
```

Is this correct? (10/10)



```
type T1 is range 1 .. 10;
procedure Proc (V : T1);

type T2 is new T1;

type T3 is new T2;
overriding procedure Proc (V : T3);
end P;
```

Everything is fine.

T2 has an implicit derivation of Proc, which is then overridden with T3.





university.adacore.com