Start

A list of topics appear on the Lecture schedule and in this document. You will need to work in a group of three students to both:

1. write notes about one of the topics
2. give a half hour presentation and answer questions about your topic.

The notes should be designed for use when revising for the final exam and will be publicly available to all students. The student groups and topics will be finalized on the Thursday of Week 1. You must present your choice of who you wish to work with and a list of two topics you would like to work on. The Final decision will be made by the course organizer.

SWEN421 Overview

Software correctness has long been overlooked in the rush for financial success but the advent of the Internet of Things has introduced the real possibility for many people to die due to soft ware errors. This has reignited an interest in correctness.

We will use the SPARK Ada language to explore how, in an industrial setting it is possible to write specifications as program contracts, build software and guarantee the software satisfies its contract. Using today's technology not everything can be effectively specified. Topics that are particularly problematic include *interaction* and *concurrency*.

In SPARK Ada prohibits the use of pointers and pointer arithmetic and provides push button static analysis that guarantees no buffer overflow, no referencing uninitialised data, plus flow analysis of your program.

The verification of programmer constructed specifications requires some skill but has been used in non-trivial industrial projects. Specification conditions can either be compiled onto runtime checks or statically analysed using automatic theorem provers.

To cope with large software projects Ada uses an Object Oriented approach. But sub Objects must be substitutable: behaviourally substitutable not just type matching asa in Java! If a Student **is a** Person then: the Student if asked to behave like a Person will behave exactly like a person.

This document is a break down of the headlines you need to learn. The details can be found in the resources provided. This document is not intended to be exhaustive. Reading and understanding [3,2,8] will help you to construct high quality code. In the lectures be prepared to ask and answer questions.

SPARK Ada Overview Week 1

This weeks work: install adacore's ide gps and work through examples in

Ada is designed to produce code that can be run on embedded systems and consequently must guarantee that:

1. no out of bounds memory lookup

2. no buffer overflow

Because of this **SPARK uses only fixed size data** and hence both Strings and Arrays must be of a fixed size. Adas type system is designed to support the static analysis of the code. The more restrictive the type system the easier it will be to guarantee your code. SPARK largely relies on automatic theorem proving. Nonetheless **it is the human that constructs the proof NOT the tool.** You must not expect a proof to emerge from your hacking around. You need to understand in great detail how you would construct a proof and use this to guide the tool. The tool only checks your thinking. Current tools cannot cope with a vast set of facts hence abstraction and information hiding becomes even more important than with Java style object oriented programming.

In Ada sub-programs can be either functions that return a value and can be used in expressions of procedures that do not return a value and can not be used in expressions. SPARK Ada programs are Ada programs with the SPARK mode set on (line 1 below).

```
1 package body stateOne with  SPARK_Mode => On  is
2   procedure addPerson (d: in out department; p:in Persons) is
3   begin
4     d.cnt := d.cnt +1;
5     d.people(d.cnt) := p;
6   end addPerson;
7 end stateOne;
```

In SPARK Ada all functions must be *Pure* - have no side effect. SPARK Ada specifications, just like code, need to broken down into component parts else they quickly become unreadable. Being Pure SPARK functions can safely be use in specifications.s

Before you attempt to prove any contracts (week 6) you should establish the Dependency between variables and the Flow of information in and out of both sub-programs and global variables. Both dependency and flow (Week 4) will help the built in provers.

```
total : Natural := 0;
procedure adding (x: in Natural) with
    Global => (In_Out => total),
    Depends =>  (total => (total , x));
```

Spark packages consist of a **body** name.adb containing executable code and a specification in a separate file name.ads. You can verify the code satisfies the specification but only the specification and not the code itself can be seen by any other code. In this sense the specification is an abstraction of the code, or the code is a refinement of the specification. Some aspects of the State may need to appear in the specification:

Ada uses object oriented design to decompose large problems into components. Ada objects are not like Java or C++ objects.

Designing SPARK Ada programs - overview of what you need to learn

To be effective in designing SPARK Ada code requires you to understand both how software can be specified and how specifications can be extended. You must make many very very small steps and you must verify/test as you go. But the decisions you make at each step require an understanding of [2, Section 11] and:

1. Design by contract [3, Section 5.2] ,[9,10]
2. Pre post conditions + loop invariants and variants [3, Section 5.5] (how to write loop invariants [3, Section 7.7])
3. The relation between the specification (in name.ads) and the implementation (in name.adb)
4. Use types and collections that are as detailed as possible
5. The package structure for code decomposition (see [3, Section 5.3] for Abstract state and Refined state and the use of child packages for necessary visibility)
6. The formal libraries provide a higher level of abstraction especially useful for specification.
7. Ada records act as the data of objects - polymorphic objects and generic objects [2, Section 5], [3, Section 5.8]
8. Object invariant - record type invariant
9. Ada inheritance - is data refinement - based on the idea that the caller of the sub-programs is unbranching at run time and the object is passive. Hence inheritance strengthens the pre-condition (Not superposition refinement). It allows class wide invariants to be defined.
10. Ghost data, sub-programs and packages can only appear in the specification [] using ghost packages as specifications can be seen in [8].

If you have the time down load the ada project in Statefull.zip - compile and then:

1. **run >SPARK>Examine File that statically finds bugs in stateone.ads**
2. **if you can fix these bugs then run >SPARK>Prove File that will show further bugs.**

Despite great advances machine checked proofs are still hard to construct. But they can help you remove many bugs and in the hands of a skilled person even guarantee the absence of bugs.

SPARK restrictions

More online documentation is for Ada than is for SPARK Ada so beware there are some restrictions ot Ada that will be checked when you select >SPARK>Examine File option or >SPARK>Compile File.

ImporTant restrictions are:

1. No goto
2. No aliasing (no access types, ...)
3. No recursion
4. functions are Pure.

Basic file structure

Main executable program contains a single procedure in testIt.adb.

```
with Ada.Text_IO;   use Ada.Text_IO;
procedure testIt is
begin
Put_Line("hellow");
end testIt;
```

Components - think objects, modules, libraries have both specification small.ads

```
package small with SPARK_Mode => On  is
type Ids is range 0..1000;
MyId : Ids :=0; --data in package
procedure setId ( i:in Ids)
  with Global => (In_Out => MyId),
       Depends => (MyId =>+ (MyId,i));
end small;
```

and code small.adb.

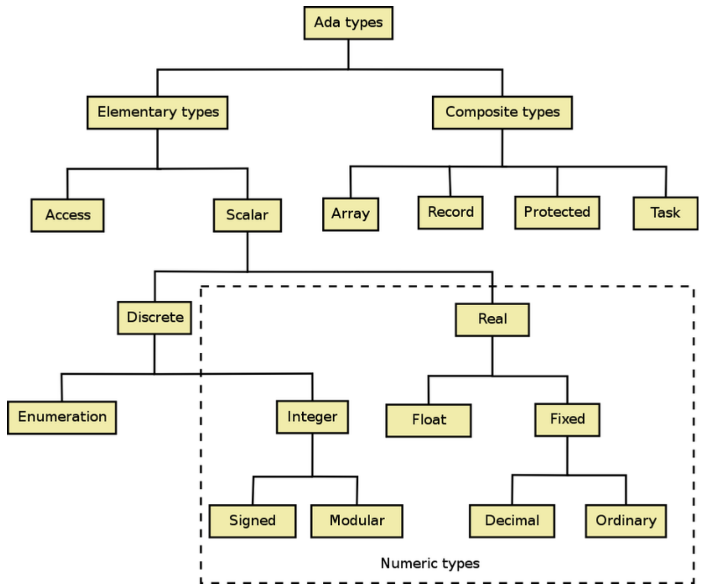
```
package body small with  SPARK_Mode => On  is
  procedure setId ( i:in Ids) is
  begin
    MyId := MyId +i;
  end setId;
end small;
```

The implementation is checked (statically verified or dynamically tested) against the specification. The implementation is private only the specification is public.

Week 2 - Types and programs -Read [3]

Statically checked and makes heavy use of enumerated types. Arrays should be indexed by enumerated types.

1. Type hierarchy:



2. type, subtype and constant

```
type RPorts is range 0..5 ;
subtype Ports is RPorts range 1..5;          -- valid Ports
invalid_Port : constant := RPorts(0);        -- invalid Port
```

]

3. Private type (predefined operations such as equality test, membership test, ... T'Base, ... are private. **The type itself is public - usable out side the package**)
4. scalar types and attributes 'First 'Last 'Image 'Value
5. discrete types and additional attributes 'Succ 'Pred
6. Real types Float and Fixed use Fixed (why?)
7. array type and attributes 'First 'Last 'Length 'Range

```
function boxPorts_toString(a: box_Ports) return String is
   s : Unbounded_String;
begin
         for i in 1..a'Length loop
            s := s & a(i)'Image ;
            if i < a'Length then
               s := s & ", ";
            end if;
         end loop;
         return To_String(s);
end boxPorts_toString;
```

8. records with Predicate =>.

```
type Packet is  record
   header : htype := invalidHeader;
```

```
    returnHeader : htype := invalidHeader;
    location : RPorts := invalid_Port;
end record
  with Predicate =>
    (if header /= invalidHeader then
         location /= invalid_Port);
```

9. object inheritance trees - tagged records *class* as sub-tree with invariances at class level
10. controlled types for customized initialisation, and finalisation, malloc ... Not allowed in SPARK. Note tis is not the same as controlling operand used in dynamic dispatch.
11. access types (pointers) Not allowed in SPARK
12. Generic types
    (a) discrete <>
    (b) integer range <>
    (c) ..

Types should be defined in a "name.ads" file and although variable can not be defined constants can and can be used to parametrise type definitions.

Sub programs

Ada has both **functions** that return a value and can be used in an expression and **procedures** that do not return a value. Expression functions are a short hand way of defining a function see **Max** and Is_Valid below.

Week 3 Packages - Read [7]

Packages are Ada's way to structure software, and can be thought of as providing libraries of different kinds.

1. Variable packages contain state that is revealed as Abstract_State in the specification and encapsulated as Refined_State in the body of the code. If tou are interested in a Class from which you only want one objects that the package is both the Class and the object.
2. Type packages do not contain state (hence neither Abstract_State nor Refined_State) but can define records that are used to model objects. The methods of these objects are sub-programs that must have the record defining the state of the object passed into it.
   Using tagged records you can define Class inheritance and hence polymorphism
3. Utility packages
4. Ghost packages

Ada packages provide data encapsulation. A package can be a singleton object. For Packages, Classes, that define multiple objects encapsulate the data in a record and pass the record into all sub programs for the object.

```
private
type Stack is record
     Content : Element_Array := (others => 0);
     Size    : My_Length := 0;
     Max     : Element := 0;
end record with
      Type_Invariant => Is_Valid (Stack);

function Is_Valid(S:Stack) return Boolean is
        ((for all I in 1..S.Size => S.Content(I) <= S.Max)
          and (if S.Max > 0 then
         (for some I in 1..S.Size => S.Content(I) = S.Max)));

function Max(S:Stack) return Element is (S.Max);
end P;
```

Week 4 Flow Read [3]

Types are statically checked by the compiler. Flow analysis can also be statically checked see menu SPARK and option Examine File. Types and Flow can be used to both find common errors and to help the proof engine.

```
1  procedure addPerson ( p:in Persons)
2    with Global => (Output => latest,
3                    In_Out => (cnt,dept)),
4          Depends => (latest =>p,
5                      cnt => cnt,
6                      dept =>+ (p,cnt));
```

The data dependency contract Global (lines 2,3 above) must be complete and defines which global variable in In, Out or both In_Out. The flow contract Depends (lines 4,5,6 above) also must be complete but, for both global variables and parameters defines which are dependent up on which variable or input parameter.

Neither parts of an array nor fields in a record can appear in these contracts.

Hiding the state of an Object from its clients while letting a set of methods be public is a standard abstraction technique. But the Flow contracts in the public specification may need to refer to the state of the Object while preserving details of the state as private. This can be achieved using Abstract_State in the specification and Refined_State in the body of the implementation.

Week 5 Proofs - Read [10,2]

Logic language in Ada.

pragmas Assert, Assume, Loop_Invariant, Perdicate, Pre, Post,

Specifications as OO Contracts

Design by contract [10,9] can be applied to SPARK Ada specifications. Important issues include:

1. Pre and Post conditions
2. Loop invariants and variants
3. Type invariants that can be used as object invariants
4. Class invariants that apply to any object that **is a** object of the class

When proofs fail

It is surprisingly common for either the code of the specification to be wrong and time taken trying to construct the proof is wasted. Remember you must understand the proof in detail. When it fails the tool will usually show you a counter example. If what you are trying to proove is correct then the counter example show you limitation of the tools understanding. You need to add assert pragmas to help it - blaming the tool for being stupid may be fun but does not help.

1. Check your **code is correct**
2. Check your **specification is correct** - not to precise nor too lax
3. Proof debugging by adding **temporary assume** statements.
   Describe why you think the statement is true adding simpler assumptions that you used. Finally, based on your assumption, the proof will work. Now you have to change the assumptions to assertions. If the proof still works keep the assertion if it fails you have the simpler assumption to prove. Note the assert statement will be checked for correctness
4. Use smaller development steps
5. Add axioms such as the transitivity of **bigger** - but try to avoid using assume!
6. Use testing to cover unproven conditions

Week 6 Objects and Inheritance Read [2]

Dynamic dispatch is permitted in SPARK Ada and is analysable because of the strict interpretation of Liskov substitution principle.

Ada inheritance:

```
type subclass_type is new superclass_type with extension_part;
```

like data refinement it preserves the operational behaviour of the super class - a faithful interpretation of the well known Liskov substitution principle. This is achieved by checking:

1. sub-class precondition is implied by the super-class precondition
2. sub-class postcondition implies super-class postcondition

Classes, in type packages, that are going to be extended using inheritance use tagged records.

```
1 type M1d is   tagged  record
2    Car_Num : CarNum := 0;
3 end record
4    with Predicate => car_num <= Max;
```

The Super class has a record structure with fields that extend the sub class record fields

```
1 type  M2d is new M1d  with record
2    Isml : CarNum := 0;
3    Mlis : CarNum := 0;
4    Isl  : CarNum := 0;
5 end record
6    with Predicate =>
7        (Car_Num = Isml+ Mlis + Isl) and
8        (Isml = 0 or Mlis =0);
```

Note the **Car_Num** field line 2 in **M1d** is implicitly in **M2d** and used in line 7.

Inheritance constructs a tree of Classes with Super Classes above, or further towards the leaves than, the Sub Classes. Specific individual Classes are referred to in specifications by their name and the set of Classes above a named Class, all Sub Classes of the named Class, is referred to by name'Class.

Using an Ada class as a parameter gives us *Polymorthic* sub-programs. The actual code used has to be decided at run time *dynamic dispatch*.

Abstract objects can not be executed

Week 7 Generics - Read [6]

Both packages and sub-programs can be generic, that is they can have parameters instantiated in different ways at compile time. The generic parameters can be types or values or functions.

Defining Swap with generic type Element_T.

```
generic
type Element_T is private;  -- Generic formal type parameter
procedure Swap (X,Y: in out Element_T);
```

```
procedure Swap (X,Y: in out Element_T) is
Temporary : constant Element_T := X;
begin
X := Y;
Y := Temporary;
end Swap;
```

Instantiating the generic type to Integer. The procedure Swap cannot be used as the type of its data is unknown but Swap_Integers can be used as all type information has been provided.

```
procedure Swap_Integers is new Swap (Integer);
```

Week 8 SPARK Formal Libraries- Read [5,4]

The Ada standard libraries include a few formal SPARK libraries [4] that include **Sets, Maps, Lists** and **Vectors**. These can be used to raise the level of abstraction in your specifications and have been used in the construction of data refinements see [8] for details.

Library details can be seen by first adding, for example,

with Ada.Formal_Doubly_Linked_Lists;

and then right clicking on "Formal_Doubly_Linked_Lists" in the editor and selecting the option go to declaration of. Alternatively an example can be found in Appendix A of this document. There are also Lemma Libraries that might help you construct proofs but, see [3, Section 5.10.2], the tool needs to be configured to use them and they are a very new addition.

Example code instantiating the generic list line1 1-3 and then instantiating the list size line 4:

```
1  package Message_Gen is new
2     Formal_Doubly_Linked_Lists(Element_Type => Packet,
3                                "="          => Equ_Packet);
4  type Message is  new  Message_Gen.List(packet_Cnt);
```

Example code using the list: note in line 2 the Cursor is defined in `Message_Gen`.

```
1  procedure print_message( s: in String; m: in Message) is
2    Cu : Message_Gen.Cursor := First (m);
3    begin
4      Put_Line(s& " Message ");
5      while Has_Element (m, Cu) loop
6          Put_Line(Packet_toString(Element(Container => m,
7                                           Position  => Cu)));
8          Next (m, Cu);
9      end loop;
10     Put_Line(" end");
11 end print_message;
```

Week 9 INFORMED design Methodology - Read [1]

Packages are Ada's way to structure software, and can be thought of as providing libraries of different kinds.

1. Variable packages contain state that is revealed as **Abstract_State** in the specification and encapsulated as **Refined_State** in the body of the code. If tou are interested in a Class from which you only want one objects that the package is both the Class and the object.
2. Type packages do not contain state (hence neither Abstract_State nor Refined_State) but can define records that are used to model objects. The methods of these objects

are sub-programs that must have the record defining the state of the object passed into it.
Using tagged records you can define Class inheritance and hence polymorphism

3. Utility packages
4. Boundary Variable packages: Input must appear in a separate package from output.
5. Boundary Variable Abstraction package: All external reference must appear in Boundary Variable package.
6. Generic Variable package. These are Variable packages with a parameter that needs to be instantiated as a side effect the package is now a Class from which any number of objects can be built. Some polymorphism can be achieved using type parameters.

Ada packages provide data encapsulation. A package can be a singleton object. For Packages, Classes, that define multiple objects encapsulate the data in a record and pass the record into all sub programs for the object.

```
private
type Stack is record
Content : Element_Array := (others => 0);
Size    : My_Length := 0;
Max     : Element := 0;
end record with
Type_Invariant => Is_Valid (Stack);

function Is_Valid(S:Stack) return Boolean is
((for all I in 1..S.Size => S.Content(I) <= S.Max)
and (if S.Max > 0 then
(for some I in 1..S.Size => S.Content(I) = S.Max)));

function Max(S:Stack) return Element is (S.Max);
end P;
```

Week 10 Further Details of Proof

References

1. Ada core INFORMED documentation SPARK 2014. `http://docs.adacore.com/sparkdocs-docs/Informed.htm`.
2. Ada Core Safe and Secure Sorftware SPARK2014 - SWEN421 home page. `http://www.embedded.com/design/programming-languages-and-tools/4433251/Safe-and-secure-object-oriented-programming-with-Ada-2012-s-contracts`.
3. Ada Core SPARK documentation SPARK 2014. `http://docs.adacore.com/spark2014-docs/html/ug/en/spark_2014.html`.
4. Formal SPARK libraries. `http://docs.adacore.com/spark2014-docs/html/ug/en/source/spark_libraries.html`.
5. Rational for Ada2005 Containers. `www.adacore.com/uploads/technical-papers/Ada05_rational_07.pdf`.

6. Wiki Books Ada Generics. https://en.wikibooks.org/wiki/Ada_Programming/Generics.
7. Wiki Books Ada Packages. https://en.wikibooks.org/wiki/Ada_Programming/Packages.
8. C. Dross and Y. Moy. *Abstract Software Specifications and Automatic Proof of Refinement*, pages 215–230. Springer International Publishing, Cham, 2016.
9. B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.
10. B. Meyer. Design by contract: Making object-oriented programs that work. In *TOOLS 1997: 25th International Conference on Technology of Object-Oriented Languages and Systems, 24-28 November 1997, Melbourne, Australia*, page 360, 1997.

Appendix A

```
----------------------------------------------------------------------------
--
--
--                         GNAT LIBRARY COMPONENTS
--
--
--
--                 ADA.CONTAINERS.FORMAL_DOUBLY_LINKED_LISTS
--
--
--
--                                 S p e c
--
--
--
--          Copyright (C) 2004-2015, Free Software Foundation, Inc.
--
--
--
-- This specification is derived from the Ada Reference Manual for use with --
-- GNAT. The copyright notice above, and the license provisions that follow --
-- apply solely to the  contents of the part following the private keyword. --
--
--
-- GNAT is free software;  you can  redistribute it  and/or modify it under --
-- terms of the  GNU General Public License as published
by the Free Soft- --
-- ware  Foundation;  either version 3,  or (at your option) any later ver- --
-- sion.  GNAT is distributed in the hope that it will be useful, but WITH- --
-- OUT ANY WARRANTY;  without even the  implied warranty of MERCHANTABILITY --
-- or FITNESS FOR A PARTICULAR PURPOSE.
--
--
--
--
--
--
--
--
--
--
```

```
-- You should have received a copy of the GNU General Public License and --
-- a copy of the GCC Runtime Library Exception along with this program; --
-- see the files COPYING3 and COPYING.RUNTIME respectively. If not, see --
-- <http://www.gnu.org/licenses/>. --
------------------------------------------------------------------------------

-- This spec is derived from Ada.Containers.Bounded_Doubly_Linked_Lists in the
-- Ada 2012 RM. The modifications are meant to facilitate formal proofs by
-- making it easier to express properties, and by making the specification of
-- this unit compatible with SPARK 2014. Note that the API of this unit may be
-- subject to incompatible changes as SPARK 2014 evolves.

-- The modifications are:

--   A parameter for the container is added to every function reading the
--   contents of a container: Next, Previous, Query_Element, Has_Element,
--   Iterate, Reverse_Iterate, Element. This change is motivated by the need
--   to have cursors which are valid on different containers (typically a
--   container C and its previous version C'Old) for expressing properties,
--   which is not possible if cursors encapsulate an access to the underlying
--   container.

--   There are three new functions:

--     function Strict_Equal (Left, Right : List) return Boolean;
--     function First_To_Previous (Container : List; Current : Cursor)
--        return List;
--     function Current_To_Last (Container : List; Current : Cursor)
--        return List;

--   See subprogram specifications that follow for details

generic
type Element_Type is private;

with function "=" (Left, Right : Element_Type)
return Boolean is <>;

package Ada.Containers.Formal_Doubly_Linked_Lists with
Pure,
SPARK_Mode
```

```
is
pragma Annotate (GNATprove, External_Axiomatization);
pragma Annotate (CodePeer, Skip_Analysis);

type List (Capacity : Count_Type) is private with
Iterable => (First       => First,
Next        => Next,
Has_Element => Has_Element,
Element     => Element),
Default_Initial_Condition => Is_Empty (List);
pragma Preelaborable_Initialization (List);

type Cursor is private;
pragma Preelaborable_Initialization (Cursor);

Empty_List : constant List;

No_Element : constant Cursor;

function "=" (Left, Right : List) return Boolean with
Global => null;

function Length (Container : List) return Count_Type with
Global => null;

function Is_Empty (Container : List) return Boolean with
Global => null;

procedure Clear (Container : in out List) with
Global => null;

procedure Assign (Target : in out List; Source : List) with
Global => null,
Pre    => Target.Capacity >= Length (Source);

function Copy (Source : List; Capacity : Count_Type := 0) return List with
Global => null,
Pre    => Capacity = 0 or else Capacity >= Source.Capacity;

function Element
(Container : List;
Position : Cursor) return Element_Type
with
Global => null,
Pre    => Has_Element (Container, Position);
```

```
procedure Replace_Element
(Container : in out List;
Position   : Cursor;
New_Item   : Element_Type)
with
Global => null,
Pre    => Has_Element (Container, Position);

procedure Move (Target : in out List; Source : in out List) with
Global => null,
Pre    => Target.Capacity >= Length (Source);

procedure Insert
(Container : in out List;
Before     : Cursor;
New_Item   : Element_Type;
Count      : Count_Type := 1)
with
Global => null,
Pre    => Length (Container) + Count <= Container.Capacity
and then (Has_Element (Container, Before)
or else Before = No_Element);

procedure Insert
(Container : in out List;
Before     : Cursor;
New_Item   : Element_Type;
Position   : out Cursor;
Count      : Count_Type := 1)
with
Global => null,
Pre    => Length (Container) + Count <= Container.Capacity
and then (Has_Element (Container, Before)
or else Before = No_Element);

procedure Insert
(Container : in out List;
Before     : Cursor;
Position   : out Cursor;
Count      : Count_Type := 1)
with
Global => null,
Pre    => Length (Container) + Count <= Container.Capacity
and then (Has_Element (Container, Before)
```

```
or else Before = No_Element);

procedure Prepend
(Container : in out List;
New_Item   : Element_Type;
Count      : Count_Type := 1)
with
Global => null,
Pre    => Length (Container) + Count <= Container.Capacity;

procedure Append
(Container : in out List;
New_Item   : Element_Type;
Count      : Count_Type := 1)
with
Global => null,
Pre    => Length (Container) + Count <= Container.Capacity;

procedure Delete
(Container : in out List;
Position   : in out Cursor;
Count      : Count_Type := 1)
with
Global => null,
Pre    => Has_Element (Container, Position);

procedure Delete_First
(Container : in out List;
Count      : Count_Type := 1)
with
Global => null;

procedure Delete_Last
(Container : in out List;
Count      : Count_Type := 1)
with
Global => null;

procedure Reverse_Elements (Container : in out List) with
Global => null;

procedure Swap
(Container : in out List;
I, J       : Cursor)
with
```

```
Global => null,
Pre    => Has_Element (Container, I) and then Has_Element (Container, J);

procedure Swap_Links
(Container : in out List;
I, J       : Cursor)
with
Global => null,
Pre    => Has_Element (Container, I) and then Has_Element (Container, J);

procedure Splice
(Target : in out List;
Before : Cursor;
Source : in out List)
with
Global => null,
Pre    => Length (Source) + Length (Target) <= Target.Capacity
and then (Has_Element (Target, Before)
or else Before = No_Element);

procedure Splice
(Target   : in out List;
Before   : Cursor;
Source   : in out List;
Position : in out Cursor)
with
Global => null,
Pre    => Length (Source) + Length (Target) <= Target.Capacity
and then (Has_Element (Target, Before)
or else Before = No_Element)
and then Has_Element (Source, Position);

procedure Splice
(Container : in out List;
Before     : Cursor;
Position   : Cursor)
with
Global => null,
Pre    => 2 * Length (Container) <= Container.Capacity
and then (Has_Element (Container, Before)
or else Before = No_Element)
and then Has_Element (Container, Position);

function First (Container : List) return Cursor with
Global => null;
```

```
function First_Element (Container : List) return Element_Type with
Global => null,
Pre    => not Is_Empty (Container);

function Last (Container : List) return Cursor with
Global => null;

function Last_Element (Container : List) return Element_Type with
Global => null,
Pre    => not Is_Empty (Container);

function Next (Container : List; Position : Cursor) return Cursor with
Global => null,
Pre    => Has_Element (Container, Position) or else Position = No_Element;

procedure Next (Container : List; Position : in out Cursor) with
Global => null,
Pre    => Has_Element (Container, Position) or else Position = No_Element;

function Previous (Container : List; Position : Cursor) return Cursor with
Global => null,
Pre    => Has_Element (Container, Position) or else Position = No_Element;

procedure Previous (Container : List; Position : in out Cursor) with
Global => null,
Pre    => Has_Element (Container, Position) or else Position = No_Element;

function Find
(Container : List;
Item      : Element_Type;
Position  : Cursor := No_Element) return Cursor
with
Global => null,
Pre    => Has_Element (Container, Position) or else Position = No_Element;

function Reverse_Find
(Container : List;
Item      : Element_Type;
Position  : Cursor := No_Element) return Cursor
with
Global => null,
Pre    => Has_Element (Container, Position) or else Position = No_Element;

function Contains
```

```
(Container : List;
Item       : Element_Type) return Boolean
with
Global => null;

function Has_Element (Container : List; Position : Cursor) return Boolean
with
Global => null;

generic
with function "<" (Left, Right : Element_Type) return Boolean is <>;
package Generic_Sorting with SPARK_Mode is

function Is_Sorted (Container : List) return Boolean with
Global => null;

procedure Sort (Container : in out List) with
Global => null;

procedure Merge (Target, Source : in out List) with
Global => null;

end Generic_Sorting;

function Strict_Equal (Left, Right : List) return Boolean with
Ghost,
Global => null;
--  Strict_Equal returns True if the containers are physically equal, i.e.
--  they are structurally equal (function "=" returns True) and that they
--  have the same set of cursors.

function First_To_Previous (Container : List; Current : Cursor) return List
with
Ghost,
Global => null,
Pre    => Has_Element (Container, Current) or else Current = No_Element;

function Current_To_Last (Container : List; Current : Cursor) return List
with
Ghost,
Global => null,
Pre    => Has_Element (Container, Current) or else Current = No_Element;
--  First_To_Previous returns a container containing all elements preceding
--  Current (excluded) in Container. Current_To_Last returns a container
--  containing all elements following Current (included) in Container.
```

```ada
--  These two new functions can be used to express invariant properties in
--  loops which iterate over containers. First_To_Previous returns the part
--  of the container already scanned and Current_To_Last the part not
--  scanned yet.

private
pragma SPARK_Mode (Off);

type Node_Type is record
Prev    : Count_Type'Base := -1;
Next    : Count_Type;
Element : Element_Type;
end record;

function "=" (L, R : Node_Type) return Boolean is abstract;

type Node_Array is array (Count_Type range <>) of Node_Type;
function "=" (L, R : Node_Array) return Boolean is abstract;

type List (Capacity : Count_Type) is record
Free   : Count_Type'Base := -1;
Length : Count_Type := 0;
First  : Count_Type := 0;
Last   : Count_Type := 0;
Nodes  : Node_Array (1 .. Capacity) := (others => <>);
end record;

type Cursor is record
Node : Count_Type := 0;
end record;

Empty_List : constant List := (0, others => <>);

No_Element : constant Cursor := (Node => 0);

end Ada.Containers.Formal_Doubly_Linked_Lists;
```