Overview of Presentation

• *Introducing... Ada 2012*

• **Ada 2012 addresses *Ongoing* Development Challenges**
  – Reducing the *gap* between design and code
  – Reducing the *cost* of testing and certification
  – Reducing the *latent* error rate

• **Ada 2012 addresses *New Challenges for 2013*+**
  – Model-based design
  – Multicore and Distributed Computing
  – Increasing use of Formal Methods

• **The Ada Tools Ecosystem**

• **Ada 2012 Coding Examples and Exercises**

• **Ada 2012 Summary**
Ada 2012: ongoing evolution of a language focused on software reliability

- **Ada 83**: strong typing, packages and information hiding, tasking
  - First mainstream language to incorporate information hiding, subtyping, concurrency and real-time constructs.

- **Ada 95**: object-orientation, dynamic dispatching, protected types
  - OOP + strong typing, hierarchical libraries, efficient synchronization tools
  - Rules to prevent dangling pointers

- **Ada 2005**: multiple inheritance of interfaces, container libraries
  - OOP + concurrency, specialized profiles for real-time programming
Highlights of Ada 2012

• **Programming by contract:**
  – Pre- and Post-conditions for subprograms
  – Type invariants
  – Subtype predicates

• **Constructs for programming multicores:**
  – Dispatching domains, processor affinities

• **Greater expressivity:**
  – Expression functions, quantified expressions, conditional expressions, iterators
Ada 2012 Programming by contract

- **Pre- and post-conditions for subprograms**
  - Call is legal if initial conditions satisfy precondition predicate
  - Subprogram works properly if result satisfies postcondition predicate

- **Type invariants:**
  - Every visible value of the type must satisfy a consistency condition
  - For private types and type extensions: specify a consistency condition that objects of the type must obey (e.g. the entries in a bar chart must add up to 100%)
  - Interacts well with OOP

- **Subtype predicates:**
  - Only a subset of the values of the type are legal values
generic
  type Item is private;
package Stack_Interfaces is
  type Stack is interface;
  function Is_Empty (S : Stack) return Boolean is abstract;
  function Is_Full (S : Stack) return Boolean is abstract;

  procedure Push (S : in out Stack; I : in Item) is abstract
    with Pre'Class => not Is_Full (S),
    Post'Class => not Is_Empty (S);

private
...

end Stack_Interfaces;
package Bars is
    type Bar_Chart is private
        with Type_Invariant => Is_Complete(Bar_Chart);
    function Is_Complete (X : Bar_Chart)  return Boolean;
private
    type Bar_Chart is array (1 .. 10) of Integer;
end Bars;

package body Bars is
    function Is_Complete (X : Bar_Chart) is
        --  verify that component values add up to 100
    end;
Contracts and Program Correctness

• Contracts help the programmer (force the programmer?) to make his intention more explicit (strong typing is an earlier step in the same direction).

• Checking of contract may be
  – static (compiler)
  – dynamic (run-time assertions)

• Contracts help develop testing protocols

• Contracts complement and assist static analysis tools

• Ada 2012 is one of the first mainstream language to incorporate contracts as a general programming tool
Ada 2012 Concurrency and real-time features

- Addresses the multicore revolution
- Memory barriers and Volatile objects
- Managing affinities on multiprocessors
- Ravenscar Profile for Multiprocessor Systems
- Task barriers
Efficient use of multicore architectures

• Builds on Ada Tasking model (higher level than thread programming in C++ or Java):
  – High level constructs: tasks and protected types
  – Defined communication patterns: rendez-vous and protected operations
  – Incorporates real-time constructs: timed operations, delays, abort

• New in Ada 2012: dispatching domains:
  – Group of computing units with common dispatching queue

• Map task to domain or to single processor
Ada 2012 and Programming expressiveness

- More powerful expression mechanisms
  - Conditional expressions
  - Expression functions
  - Quantified expressions

- Comprehensive iterators on arrays and containers
Ada 2012 Quantified expressions

State that A is sorted:

(for all J in A'First .. T'Pred (A'Last) =>
 A (J) <= A (T'Succ (J)))

State that N is not a prime number:

(for some X in 2 .. N / 2 =>
 N mod X = 0)

*some* is a new reserved word
Ada 2012 Container/Array Iterators

- Allows indexing over containers, with and without cursors:

```ada
for Cursor in Iterate (Container) loop
    Container (Cursor) := Container (Cursor) + 1;
end loop;
```

```ada
for Thing of Box loop
    Modify (Thing);
end loop;
```

Both forms apply to arrays and containers.
Ada 2012 for Ongoing Development Challenges

- **Reducing the *gap* between design and code**
  - Ada 83 provided a portable, high-level message-passing-based tasking model;
  - Ada 95 added data-oriented synchronization via an elegant protected type structure, a safe and efficient follow-on to Hoare and Per Brinch Hansen’s *conditional critical regions* and *monitors*;
  - Ada 2005 added synchronized type hierarchies represented using interfaces, parameterized when appropriate as generic templates;
  - Ada 2012 allows requirements to be represented directly in the code as preconditions, postconditions, type invariants, and named predicates.

*Ada still provides lower-level control:*
- Precise record layout specifications at the bit level
- CPU affinity specifications for multicore
- Subpool specifications for efficient storage management
Reducing the Gap between Design and Code

- **Requirements and Design**
  - Ada’s Portable High-Level Features
  - Ada’s Lower-Level Control
  - Other Languages

- **Machine level**

Bigger gaps
Ada 2012 for Ongoing Development Challenges (cont’d)

- **Reducing the cost of testing and certification**
  - Ada provides a *very strong* type model
    - Including strongly differentiated numeric and enumeration types
    - In many other languages, almost everything is effectively an “int”
  - Ada does more *checking* at compile-time
    - Enforces type matching for array *indexing*
    - Enforces strict *matching* of constraints and non-nullness
    - Checks for potentially *dangling* pointers
    - Checks *in*, vs. *out*, vs. *in-out* parameters
  - Ada backs up compile-time checking with run-time checks as needed
    - Range or numeric *overflow*
    - Assertions, preconditions, postconditions, *invariants*, etc.
    - Task *priority ceilings*
    - Can be *suppressed* if desired after testing is complete
  - Ada and third-party tools support *formal verification* when needed
Ada 2012 for Ongoing Development Challenges (cont’d)

- Reducing *latent* errors in released code
  - Ada compiler, run-time, and additional tools form a *gauntlet* that only the *logically consistent* program can survive;
  - Error rate in delivered Ada code is significantly lower than that in other less stringent languages;
  - Programs written in provable subset and analyzed formally, such as SPARK and the SPARK tool set, have approached zero latent errors in delivered code.

---

*The Ada Gauntlet*

*each check is simple but only the logical survive*
Ada 2012 for the New Challenges of 2013+

• **Model-based Design**
  
  – Ada as the *target* compilable language *generated* from a Simulink, SCADE, or UML model
    
    • benefits from Ada’s portable high-level constructs which can match more closely the modeling constructs, reducing the *gap* between the model and the implementation;
  
  – Ada as the language for the *hand-written* components
    
    • benefits from Ada’s higher-level features also, because it allows the run-time models between the *generated* and *hand-written* components to be consistent at a higher level, with, for example, a shared tasking and synchronization model, and a shared strongly-typed structure;
  
  – Ada provides an *extra* level of *checking*
    
    • on the *generated* code,
    
    • on the interfaces *between* the generated code and the hand-written code,
    
    • generator can use Ada 2012 preconditions, postconditions, etc. to further strengthen the checks
Ada 2012 for New Challenges (cont’d)

- **Multicore and Distributed Computing**
  - Since the early 80’s, Ada has supported multitasking;
  - Ada 2012 adds lower-level control over:
    - CPU *affinities* to individual cores, or to dispatching *domains*
    - CPU time *budgets* for individual tasks and groups of tasks
  - Ada has a standard distributed computing model
    - Supports *statically-bound* remote procedure calls;
    - Supports *dynamically-bound* remote object-oriented dispatching calls;
    - Supports *shared data* and *shared nothing* models;
    - Allows user control over *marshalling* and *communication* subsystem.
  - Ada tools vendors also support Ada in heterogeneous systems
    - Real-Time CORBA
    - DDS (OMG’s Distributed Data Service)

*Portable higher-level model with lower-level control*
Ada 2012 for New Challenges (cont’d)

• Increasing use of formal methods
  – Ada has formally-defined numeric models
    • strongly-differentiated numeric and enumeration types
    • specifiable range constraints on all scalar types
  – Ada has formally-defined task scheduling models
    • FIFO within priorities, Earliest Deadline First, etc.
  – Ada has formally-defined data sharing models
    • Independent, Volatile, Atomic, Protected, Ceiling Priority
  – Ada 2012 adds:
    • Preconditions, Postconditions,
    • Type invariants, Named value predicates
  – Third party tools:
    • Advanced static analysis
    • Proof tools
    • Integrated proof and testing tools
Ada Tools Ecosystem

- **Modeling Tools from:**
  - Atego, Ellidis, Esterel, IBM TeleLogic, MathWorks and others

- **Compilers from:**
  - AdaCore, Atego (including new Apex acquisition), DDC-I, Green Hills, Irvine, OC Systems and others

- **Certified RTOSs:**
  - Integrity, LynxOS, VxWorks and others

- **Static Analysis Tools from:**
  - AdaCore, Grammatech, LDRA, MathWorks and others

- **Testing Tools from:**
  - Compiler vendors, plus LDRA, Vector S/W, IPL and others

- **Traceability Tools from:**
  - Atego, IBM TeleLogic, LDRA, VeroCel, and others
Digging Into Ada 2012

Coding Examples and Exercises
generic
type Item is private;
package Stack_Interfaces is
type Stack is interface;
function Is_Empty (S : Stack) return Boolean is abstract;
function Is_Full (S : Stack) return Boolean is abstract;
procedure Push (S : in out Stack; I : in Item) is abstract;
function Pop (S : in out Stack) return Item is abstract;
end Stack_Interfaces;
generic
package Stack_Interfaces.Bounded is
  type Bounded_Stack(<> ) is new Stack with private;
  function Create(Size: Natural) return Bounded_Stack;

  function Size(S : Bounded_Stack) return Natural;
  function Count(S : Bounded_Stack) return Natural;

  function Is_Empty (S : Bounded_Stack) return Boolean
      is (Count(S) = 0); -- expression functions
  function Is_Full (S : Stack) return Boolean
      is (Count(S) = Size(S)); -- expression functions

  procedure Push (S : in out Bounded_Stack; I : in Item);

  function Pop(S : in out Bounded_Stack) return Item;

private ...
Bounded Stack Internals

generic
package Stack_Interfaces.Bounded is

private
  type Item_Array is array(Positive range <> ) of Item;
  type Bounded_Stack(Size : Natural) is new Stack with record
    Count : Natural := 0;
    Data : Item_Array(1..Size);
  end record;
end Stack_Interfaces.Bounded;

package body Stack_Interfaces.Bounded is

procedure Push (S : in out Bounded_Stack; I : in Item) is
begin
  S.Count := S.Count + 1;
  S.Data(S.Count) := I;
end Push;
end Stack_Interfaces.Bounded;
What sort of Pre- and Postconditions are appropriate here?

- **Preconditions** prevent failures; **Postconditions** define effects
- Push will get an index out of bounds if S.Count = S.Size on entry
- Create precondition to prevent that:
  
  ```ada
  procedure Push(…) with Pre => Count(S) < Size(S);
  ```

- Now we have the following code:
  ```ada
  Stk : BI_Inst.Bounded_Stack := BI_Inst.Create(10);
  ...
  BI_Inst.Push(Stk, X); -- Can we be sure this will satisfy the Pre?
  ```

- We need a **Post** on Create to know initial Size and Count:
  ```ada
  function Create(...) return Bounded_Stack
  with Post => Bounded.Size(Create’Result) = Size
  and Count(Create’Result) = 0;
  ```

- We also need a **Post** on Push itself so 10 Pushes are known safe:
  ```ada
  procedure Push(…) with Pre => Count(S) < Size(S),
  Post => Count(S) = Count(S)’Old + 1;
  ```
Bounded Stack with Pre/Postconditions

generic
package Stack_Interfaces.Bounded is
  type Bounded_Scal (<>) is new Stack with private;
  function Create(Size: Natural) return Bounded_Scal
    with Post => Bounded_Size(Create'Result) = Size
              and Count(Create'Result) = 0;
  function Size(S : Bounded_Scal) return Natural;
  function Count(S : Bounded_Scal) return Natural
    with Post => (Count(S) <= Size(S));
  function Is_Empty (S : Bounded_Scal) return Boolean
    is (Count(S) = 0);
  function Is_Full (S : Stack) return Boolean
    is (Count(S) = Size(S));

  procedure Push (S : in out Bounded_Scal; I : in Item)
    with Pre => Count(S) < Size(S),
              Post => Count(S) = Count(S)'Old + 1;
  function Pop(S : in out Bounded_Scal) return Item
    with Pre => Count(S) > 0,
              Post => Count(S) = Count(S)'Old - 1;

private ...

Now suppose we use the abstract stack...

- Imagine we have a class-wide operation:
  ```ada
  procedure Replace_Top(S : in out Stack'Class; I : Item) is
    Discard : constant Item := Pop(S);
  begin
    Push(S, I);
  end Replace_Top;
  ```

- Need a classwide precondition on Pop, and a normal precondition on Replace_Top to make things safe:
  ```ada
  function Pop(...) with Pre'Class => not Is_Empty(S)
  procedure Replace_Top(...) with Pre => not Is_Empty(S);
  ```

- Need a classwide postcondition on Push and a normal postcondition on Replace_Top to safely do it twice:
  ```ada
  procedure Push(...) with Post'Class => not Is_Empty(S)
  procedure Replace_Top(...) with Post => not Is_Empty(S)
  ```

- Classwide pre/postconds must be checked on overridings
generic
type Item is private;
package Stack_Interfaces is
type Stack is interface;
function Is_Empty (S : Stack) return Boolean is abstract;
function Is_Full (S : Stack) return Boolean is abstract;

procedure Push (S : in out Stack; I : in Item) is abstract
with Pre'Class => not Is_Full (S),
    Post'Class => not Is_Empty (S);
function Pop (S : in out Stack) return Item is abstract
with Pre'Class => not Is_Empty (S),
    Post'Class => not Is_Full (S);
end Stack_Interfaces;
Now should verify that Bounded_Stack will abide by ancestor’s Pre’Class and Post’Class

- **Ancestor type Stack specifies:**
  procedure Push (S : in out Bounded_Stack; I : in Item)
    with Pre'Class => not Is_Full (S),
    Post'Class => not Is_Empty (S);

- **Bounded_Stack explicitly specifies:**
  function Is_Empty (S : Bounded_Stack) return Boolean
    is (Count(S) = 0);   -- not Is_Empty == Count(S) /= 0
  function Is_Full (S : Stack) return Boolean
    is (Count(S) = Size(S)); -- not Is_Full == Count(S) /= Size(S)
  procedure Push (S : in out Bounded_Stack; I : in Item)
    with Pre => Count(S) < Size(S),
    Post => Count(S) = Count(S)'Old + 1;

- **Liskov Substitution Principle (LSP) says:**
  - Caller sees ancestor precondition, so must *imply* descendant precondition
  - Caller sees ancestor postcondition, so must *be implied by* descendant postcondition
  - Verified:
    Count(S) /= Size(S) and Count(S) <= Size(S) ⇒ Count(S) < Size(S)
    Count(S) = Count(S)’Old+1 and Count(S)’Old >= 0 ⇒ Count(S) /= 0
Ada 2012 and Liskov Substitution Principle

- Ada 2012 compiler is *not* required to statically check that Pre’Class implies Pre nor that Post implies Post’Class
  - Ada 2012 compiler is only required to do run-time checks
  - Other tools can attempt proofs that the run-time checks will not fail

- Ada 2012 language ensures implications by *effectively*:
  - “or”ing Pre’Class of ancestors with Pre’Class of descendant, and
  - “and”ing Post’Class of ancestors with Post’Class of descendant

- The Pre’Class “or”ing is done “implicitly”:
  - In a “dispatching” call, caller only checks the Pre’Class annotations that they can “see”;
  - Pre’Class of descendants of T where controlling operand is of type T’Class are *not* even checked.

- The Post’Class “and”ing is done by checking all of them.
package Bars is
  type Bar_Chart is private
      with Type_Invariant => Is_Complete(Bar_Chart);
  function Is_Complete (X : Bar_Chart) return Boolean;
private
  type Bar_Chart is array (1 .. 10) of Integer;
end Bars;

package body Bars is
  function Is_Complete (X : Bar_Chart) is
    -- verify that component values add up to 100
  end;

The Role of Type Invariants

- Type invariants are used to encode some property that is preserved by all operations on a type.
  - Becomes implicit Pre and Post condition for every operation
- Type invariants are generally introduced when attempts to prove that a given postcondition is satisfied requires that all operations guarantee certain minimum requirements.
- Example:
  - Imagine a stack of pointers, and we ensure that Push is only passed not null pointers.
  - Can we ensure that Pop returns only not null values back?
  - Solution is to come up with a Type_Invariant that says:
    - All elements at or “below” the stack pointer are /= null
    - Then show that Push (and other ops) preserve it.
  - Note that type invariants are often representation specific
    - In Ada 2012, they can be given in the private part.
generic
type T<> is limited private;
type T_Ptr is access T;
package Pointer_Stacks is
  type PointerStackTrace is private;
  procedure Push(PS : in out PointerStackTrace; Ptr : not null T_Ptr);
  function Pop(PS : in out PointerStackTrace) return not null T_Ptr;
private
  type Ptr_Array is array(Positive range <>) of T_Ptr;
type PointerStackTrace(Size : Natural) is record
    Count : Natural := 0;
    Data : Ptr_Array(1..Size) := (others => null);
  end record
  with Type_Invariant =>
    (for all I in 1..PointerStackTrace.Count =>
      PointerStackTrace.Data(I) /= null);
end Pointer_Stacks;
type Pointer_Stack(Size : Natural) is record
  Count : Natural := 0;
  Data : Ptr_Array(1..Size) := (others => null);
end record
with Type_Invariant =>
  (for all I in 1..Pointer_Stack.Count =>
    Pointer_Stack.Data(I) /= null);
end Pointer_Stacks;
package body Pointer_Stacks is
  procedure Push(PS : in out Pointer_Stack; Ptr : not null T_Ptr) is
    begin
      PS.Count := PS.Count + 1;  PS.Data(PS.Count) := Ptr;
    end Push;
  function Pop(PS : in out Pointer_Stack) return not null T_Ptr is
    begin
      PS.Count := PS.Count – 1;  return PS.Data(PS.Count + 1);
    end Pop;
end Pointer_Stacks;
Subtype Predicates
Static_Predicate and Dynamic_Predicate

• A subtype “predicate” is a generalization of the notion of a “constraint”
  – It identifies a subset of the values of a type or subtype

• Examples of constraints:
  – subtype Digits is Integer range 0..9
    – “range 0..9” is a range constraint
  – Data : Ptr_Array(1..Size)
    – “(1..Size)” is an index constraint

• Examples of predicates:
  – subtype Long_Weekend is Weekday
    with Static_Predicate =>
    Long_Weekend in Friday | Saturday | Sunday | Monday;
  – subtype Operator_Node is Node
    with Dynamic_Predicate =>
    Operator_Node.Kind in Unary_Kind | Binary_Kind;
Static vs. Dynamic Predicates

- **Static Predicate:**
  - Must apply to a scalar or string type and may involve one or more comparisons between the value being tested and static values
  - All possible values can be determined statically
  - Subtypes with such a predicate can be used as the choice in a case statement or the bounds of a loop iteration
  - Initialized objects to which such a predicate applies always satisfy the predicate

- **Dynamic Predicate:**
  - Defined by an arbitrary boolean expression involving the value being tested
  - All possible values need not be determinable statically
  - Subtypes with such a predicate can be used to declare an object and in a membership test, but may not be used for looping or as choices in a case statement
  - Some violations of the predicate might not be immediately detected
    - Only checked on certain “whole object” operations
Ada 2012 Container/Array Iterators

- Allows indexing over containers, with and without cursors:

```ada
for Cursor in Iterate (Container) loop
   Container (Cursor) := Container (Cursor) + 1;
end loop;

for Thing of Box loop
   Modify (Thing);
end loop;
```

Both forms apply to arrays and containers.
Ada 2012 Quantified expressions

State that A is sorted:

(for all J in A'First .. T'Pred (A'Last) =>
 A (J) <= A (T'Succ (J)))

State that N is not a prime number:

(for some X in 2 .. N / 2 =>
 N mod X = 0)

some is a new reserved word
Ada 2012 Summary

- Substantial enhancements over Ada 2005
- Well-defined semantics of new constructs
- Addresses high-level programming issues as well as low-level hardware concerns
- Well suited for high-reliability applications
- Brings program verification into mainstream programming

- Addresses ongoing and new challenges for developers
- Reduces costs and time-to-market while increasing quality
Ada 2012 Resources

• Everything Ada 2012:
  • http://www.ada2012.org

• Ada 2012 Reference Manual:
  – Available in various formats
  – http://www.adaic.org/ada-resources/standards/ada12/

• Annotated Ada 2012 Reference Manual:
  – (detailed presentation of design decisions)

• Technical overview: Articles by John Barnes at:
  – http://www.adacore.com/adaanswers/about/ada-2012/

• Rationale for Ada 2012: Introduction, aspects, expressions
HILT 2013 conference in Pittsburgh in November
High Integrity Language Technology

• **Keynotes and Invited Talks by:**
  – John Goodenough (SEI), Ed Clarke (CMU), Jeannette Wing (Microsoft Research), Michael Whalen (Univ. of Minnesota)

• **HILT 2013 Call for papers open until late June 2013**

• **Chance to learn more about Ada 2012**
  – Tutorials expected on Ada 2012 and high-integrity programming

• **Chance to meet companies that support the Ada Tools Ecosystem**
  – Modeling tool vendors
  – Compiler vendors
  – Static analysis and testing tool vendors

• **http://www.sigada.org/conf/hilt2013 -- November 14-20**