Overview

Pascal Leroy, IBM Rational Software
Ada is Alive and Evolving

- Ada 83 Mantra: “no subsets, no supersets”
- Ada 95 Mantra: “portable power to the programmer”
- Ada 2005 Mantra: “putting it all together”...
  - Safety and portability of Java
  - Efficiency and flexibility of C/C++
  - Unrivaled standardized support for real-time and high-integrity system
Ada is Well Supported

- Four major Ada compiler vendors
  - ACT (GNAT Pro)
  - Aonix (ObjectAda)
  - Green Hills (AdaMulti)
  - IBM Rational (Apex)

- Several smaller Ada compiler vendors
  - DDC-I, Irvine Compiler, OCSystems, RR Software, SofCheck

- Many tool vendors supporting Ada
  - IPL, Vector, LDRA, PolySpace, Grammatech, Praxis, …
ISO WG9 and Ada Rapporteur Group

- Stewards of Ada’s standardization and evolution
- Includes users, vendors, and language lawyers
  - Supported by AdaEurope and SIGAda
- First official Corrigendum released in 2001
- First language Amendment set for Fall 2005
- WG9 established overall direction for Amendment
Overall Goals for Ada 2005 Amendment

- Enhance Ada’s position as a:
  - Safe
  - High performance
  - Flexible
  - Portable
  - Interoperable
  - Concurrent, real-time, object-oriented programming language

- Further integrate and enhance the object-oriented capabilities of Ada
Ada 2005: Putting It All Together

Space and time efficiency

- Multiple interface inheritance
- Default static binding

Safety
Portability
Interoperability

Full object-orientation

Active and passive synchronized interfaces

Hard and soft real-time

EDF scheduling
Building blocks
Safety First

- The premier language for safety critical software
- Ada’s safety features are critical to making Ada a high-productivity language in all domains
- Amendments carefully designed so as to not open any safety holes
- Several amendments provide even more safety, more opportunities for catching mistakes at compile-time
Portability

- Additions to predefined Ada 95 library
  - Standard package for files and directories
  - Standard packages for calendar arithmetic, timezones, and I/O
  - Standard packages for linear algebra
  - Standard package for environment variables
  - Standard packages for containers and sorting

- Additions for real-time and high-integrity systems
  - Earliest-deadline first (EDF) and round-robin scheduling
  - Ravenscar high-integrity run-time profile
Interoperability

- Support notion of interface as used in Java, CORBA, C#, etc.
  - Interface types
  - Active and passive synchronized interface types integrate O-O programming with real-time programming

- Familiar Object.Operation notation supported
  - Uniformity between synchronized and unsynchronized types

- Support cyclic dependence between types in different packages

- Pragma Unchecked_Union for interoperating with C/C++ libraries
Technical Presentations

- Object-oriented programming
  - S. Tucker Taft
- Access types
  - John Barnes
- Structure control and limited types
  - Pascal Leroy
- Real-time improvements
  - Alan Burns
- Library stuff
  - John Barnes
- Safety
  - S. Tucker Taft
Object-Oriented Programming in Ada 2005

S. Tucker Taft, SofCheck Inc.
Overview

- Rounding out the O-O Capabilities
  - Interfaces
  - Object.Operation Notation
  - Nested Extension
  - Generic Constructor
Multiple Inheritance via Interface Types

```ada
type NT is new T and Int1 and Int2 with record
  end record;
```

- Int1 and Int2 are “interfaces”
  - Declared as: `type Int1 is interface;`
  - Similar to `abstract tagged null record` (no data)
  - All primitives must be abstract or null

- NT must provide primitives that match all primitives of Int1 and Int2
  - In other words, NT `implements` Int1 and Int2

- NT is implicitly convertible to Int1’Class and Int2’Class, and explicitly convertible back
  - and as part of dispatching, of course

- Membership test can be used to check before converting back (narrowing)
Example of Interface Types

limited with Observed_Objects;
package Observers is -- "Observer" pattern

type Observer is interface;
type Observer_Ptr is access all Observer’Class;

procedure Notify
(O : in out Observer;
Obj : access Observed_Objects.Observed_Obj’Class)
is abstract;
procedure Set_Next(O : in out Observer;
Next : Observer_Ptr) is abstract;
function Next(O : Observer) return Observer_Ptr is abstract;

type Observer_List is private;
procedure Add_Observer(List : in out Observer_List;
O : Observer_Ptr);
procedure Remove_Observer(List : in out Observer_List;
O : Observer_Ptr);
function First_Observer(List : in Observer_List)
return Observer_Ptr;
Example of Interface (cont’d)

with Observers;
with Observed_Objects;
with Graphics;
package Display3D is -- Three-dim display package.

    type View is new Graphics.Drawing3D and Observers.Observer
    and Observed_Objects.Observed_Obj with private;

    -- Must override the ops inherited from each interface.
    procedure Notify
        (V : in out View;
         Obj : access Observed_Objects.Observed_Obj’Class);
    procedure Set_Next(V : in out View;
                         Next : Observers.Observer_Ptr);
    function Next(V : View) return Observers.Observer_Ptr;

    not overriding -- This is a new primitive op.
    procedure Add_Observer_List(V : in out View;
                                 List : Observers.Observer_list);
Synchronized Interfaces

- Interface concept generalized to apply to protected and task types
- “Limited” interface can be implemented by:
  - Limited or non-limited tagged type or interface
  - Synchronized interface
- “Synchronized” interface can be implemented by:
  - Task interfaces or types ("active")
  - Protected interfaces or types ("passive")
Example of Synchronized Interfaces

- Example of protected object interface implementing (extending) a synchronized interface

```ada
type Buffer is synchronized interface;
procedure Put(Buf : in out Buffer;
  Item : in Element) is abstract;
procedure Get(Buf : in out Buffer;
  Item : out Element) is abstract;

protected type Mailbox(Capacity : Natural) is new Buffer with
  entry Put(Item : in Element);
  entry Get(Item : out Element);
private
  Box_State : ...
end Mailbox;
```
Example of Synchronized Interfaces (cont’d)

- Example of task interface implementing (extending) a synchronized interface

```ada
type Active_Buffer is task interface and Buffer;
procedure Put(Buf : in out Active_Buffer;
     Item : in Element) is abstract;
procedure Get(Buf : in out Active_Buffer;
     Item : out Element) is abstract;
procedure Set_Capacity(Buf : in out Active_Buffer;
     Capacity : in Natural) is abstract;
```

- Example of task type implementing a task interface

```ada
task type Postal_Agent is new Active_Buffer with
  entry Put(Item : in Element);
  entry Get(Item : out Element);
  entry Set_Capacity(Bag_Capacity : in Natural);
  entry Send_Home_Early;  -- An extra operation.
end Postal_Agent;
```
Interfaces and Null Procedures

- No bodies permitted for primitive operations of interfaces
  - Must specify either “is abstract” or “is null”
  - This rule eliminates much of complexity of multiple inheritance

- Declaring procedure as “is null” is new in Ada 2005

- Useful for declaring a “hook” or a “call-out” which defaults to a no-op
Interfaces and Null Procedures (cont’d)

- May be used to specify:
  - A primitive procedure of a tagged type or interface, e.g.:
    ```ada
    procedure Finalize(Obj : in out Controlled) is null;
    ```
  - As default for formal procedure of a generic, e.g.:
    ```ada
    generic
        with procedure Pre_Action_Expr(E : Expr) is null;
        with procedure Post_Action_Expr(E : Expr) is null;
        with procedure Pre_Action_Decl(D : Decl) is null;
    ...
    package Tree_Walker is
    ```
Object.Operation Syntax

- More familiar to users of other object-oriented languages
- Reduces need for extensive utilization of “use” clause
- Allows for uniform reference to dispatching operations and class-wide operations, on pointers or objects
Example of Object.Operation Syntax

```ada
package Windows is
  type Root_Window is abstract tagged private;
  procedure Notify_Observers(Win : Root_Window'Class);
  procedure Display(Win : Root_Window) is abstract;
  ...
end Windows;

package Borders is
  type Bordered_Window is new Windows.Root_Window with private;
  procedure Display(Win : Bordered_Window);
  ...
end Borders;

procedure P(BW : access Bordered_Window'Class) is
begin
  BW.Display;   -- Both of
  BW.Notify_Observers; -- these “work”.
end P;
```
Nested Type Extensions

- Ada 95 requires type extension to be at same “accessibility level” as its parent type
  - i.e., cannot extend a type in a nested scope

- Ada 2005 relaxes this rule
  - Can extend inside a subprogram, task, protected, or generic body
  - Still may not extend formal type inside generic body because of possible contract violations
    - Actual type might have additional operations requiring overriding
    - Checking performed on function return and allocators
      - May not create heap object or function result that might outlive type extension

- Enables instantiation of generic containers in nested scopes, even if they use finalization, streams, or storage pools
Access Types

*John Barnes of Anonymous Access, UK*
“Playing with pointers is like playing with fire. Fire is perhaps the most important tool known to man. Carefully used, fire brings enormous benefits; but when fire gets out of control, disaster strikes.”

- Uncontrolled pointers can similarly rampage through your program
- Ada access types are nice and safe
- But Ada 95 is perhaps too rigid
  - Too many conversions
- Ada 2005 is more flexible but keeps the security
Overview

- More anonymous access types
  - Not just as access parameters (and discriminants)
- Constant and null control
  - More uniform rules
- Anonymous access to subprogram types
  - For downward closures etc
Recap 95

- All access types are named except for access parameters

```ada
type Animal is tagged record
   Legs : Integer;
   ...
end record;

type Acc_Animal is access Animal; -- Named.

procedure P(Beast : access Animal); -- Anonymous.
```
95 Constant and Null

**Named**
- Can be constant or variable
  - `access T`
  - `access constant T`
  - `access all T`
- Have `null` as a value

**Anonymous**
- Can only be variable
  - `access T`
  - `-- implies all`
- Do not have `null` as a value

Not exactly orthogonal
Not Null Everywhere

```ada
type Acc_Animal is not null access all Animal'Class;

-- An Acc_Animal must not be null and so must be initialized
-- (otherwise Constraint_Error).

type Pig is new Animal with ... ;
Empress_of_Blandings : aliased Pig := ... ;

My_Animal : Acc_Animal := Empress_Of_Blandings'Access;
```
Null Exclusion

- Advantage of null exclusion is that no check is needed on a dereference to ensure that the value is not null

- So

```
Number_Of_Legs : Integer := My_Animal.Legs;
```

is faster
Constant & Null in Access Parameters

- We can write all of the following

1. `procedure P(Beast : access Animal);`
2. `procedure P(Beast : access constant Animal);`
3. `procedure P(Beast : access all Animal);`
4. `procedure P(Beast : not null access Animal);`
5. `procedure P(Beast : not null access constant Animal);`
6. `procedure P(Beast : not null access all Animal);`

- Note that 1 and 3 are the same (compatibility)
Anonymous Access Types

- As well as in
  - access parameters
  - access discriminants

- In 2005 we can also use anonymous access types for
  - components of arrays and records
  - renaming
  - function return types

- but not for scalar variables (potential accessibility problem)
As Array Components

type Horse is new Animal with ... ;

type Acc_Horse is access all Horse'Class;
type Acc_Pig is access all Pig;

Napoleon, Snowball : Acc_Pig := ... ;
Boxer, Clover : Acc_Horse := ... ;

Animal_Farm: constant array (Positive range <>) of
  access Animal'Class :=
  (Napoleon, Snowball, Boxer, Clover);
As Record Components

type Noahs_Ark is
  record
    Stallion, Mare : access Horse;
    Boar, Sow : access Pig;
    Cockerel, Hen : access Chicken;
    Ram, Ewe : access Sheep;
  end record;

- But surely Noah took actual animals into the Ark and not just their addresses…
Linked List

- Can now write
  
  ```ada
  type Cell is
  record
      Next : access Cell;
      Value : Integer;
  end record;
  ```

- No need for incomplete declaration

- Current instance rule changed to permit this
For Function Result

- Can also declare

```ada
function Mate_Of(A : access Animal'Class)
  return access Animal'Class;
```

- We can then have

```ada
if Mate_Of(Noahs_Ark.Ewe) /= Noahs_Ark.Ram then
  -- Better get Noah to sort things out!
end if;
```
Type Conversions

- We do not need explicit conversion to anonymous types
  - They have no name anyway

- Most access type declarations are as components, few are scalar variables
  - So most objects can be of anonymous type

- This means fewer explicit conversions in OO programs
Access to Subprogram

- Remember Tinman?

- Ada 83 had no requirement for subprograms as parameters of subprograms

- Considered unpredictable since subprogram not known statically

- We were told to use generics
  - It will be good for you
  - And implementers enjoy generic sharing
Ada 95 Introduced...

- Simple access to subprogram types

```ada
type Integrand is access function(X : Float) return Float;

function Integrate(Fn : Integrand; Lo, Hi : Float) return Float;
```

- To integrate $\sqrt{x}$ between 0 and 1 we have

```ada
Result := Integrate(Sqrt'Access, 0.0, 1.0);
```

- Works OK for simple functions at library level
Problem

- But suppose we want to do
  \[ \int_0^1 \int_0^1 xy \, dx \, dy \]
- That is do a double integral where the thing to be integrated is itself an integral
- We can try…
with Integrate;  
procedure Main is

  function G(X : Float) return Float is
    function F(Y : Float) return Float is -- F is nested in G.
      begin
        return X*Y; -- Uses parameter X of G.
      end F;
    begin
      return Integrate(F'Access, 0.0, 1.0); -- Illegal in 95.
    end G;

  Result: Float;
  begin
    Result := Integrate(G'Access, 0.0, 1.00; -- Illegal in 95.
      ... 
    end Main;
Cannot Do It

- Accessibility problem

- We cannot take 'Access of a subprogram at an inner level to the access type
  - The access type Integrand is at library level
  - G is internal to Main and F is internal to G

- We could move G to library level but F has to be internal to G because F uses the parameter X of G
Anon Access to Subprogram

- Ada 2005 has anonymous access to subprogram types similar to anonymous access to object types

- The function Integrate becomes

```ada
function Integrate
  (Fn : access function (X : Float) return Float;
   Lo, Hi : Float) return Float;
```

- The parameter Fn is of anonymous type

- It now all works
Embedded Profile

function Integrate
(Fn : access function (X : Float) return Float;
Lo, Hi : Float) return Float;

- Note how the profile for the anonymous type is given within the profile for Integrate

- No problem
Other Uses

- Access to subprogram types also useful for
  - Searching
  - Sorting
  - Iterating

- Examples later in Container library
Not Null, etc.

- Access to subprogram types can also have all the exciting things that apply to access to object types

  not null, constant

- Anonymous access to subprograms as components, renaming, etc.

- Also access protected...
  - not null access protected procedure(...)
  - in Real-Time Systems annex
Conclusions

- Access type are more flexible than ever before
  - But still safe

- Access to subprogram types enable algorithms parameterized by subprograms to be written without the generic sledgehammer
Structure Control and Limited Types

Pascal Leroy, IBM Rational Software
Overview

- Multi-package type structures
  - Access to private units in private parts
  - Instantiating generics with private types
  - Partial parameter lists for formal instantiations
  - Making limited types useful
Visibility and Program Structure

- Huge changes with respect to visibility in Ada 95
- Introduction of hierarchical library units
  - Public and private children
- Intended to support large-scale structuring with enough flexibility for all application needs
- … but one problem has remained…
Multi-Package Cyclic Type Structures

- Impossible to declare cyclic type structures across library package boundaries
  - Each type must be compiled before the types that depend upon it!

- Problem existed in Ada 83, but more prominent in Ada 95

- Hierarchical library units and tagged types favor a model where each library unit represents one abstraction of the problem domain

- Workarounds are awkward
  - Mutually-dependent types have to be lumped in a single library unit...
  - ... or unchecked programming has to be used
The Cyclic Type Conundrum

with Department;
package Employee is
type Object is tagged private;
procedure Assign_Employee (Who: in out Employee.Object; To_Department: in out Department.Object);
private
type Object is tagged
record
  Assigned_To: access Department.Object;
end record;
end Employee;

with Employee;
package Department is
type Object is tagged private;
procedure Choose_Manager (For_Department: in out Department.Object; Who: in out Employee.Object);
private
type Object is tagged
record
  Manager: access Employee.Object;
end record;
end Department;

Illegal circularity!
Solution: Limited With Clauses

- Gives visibility to a *limited view* of a package
  - Contains only types and nested packages
  - Types behave as if they were incomplete
  - Cycles are permitted among limited with clauses
  - Imply some kind of “peeking” before compiling a package

- Tagged incomplete type
  - Incomplete type whose completion must be tagged
  - May be used as parameter and as prefix of ‘Class’

- No syntax for declaring a limited view: implicitly created by the compiler
Example of a Limited View

```ada
package Department is
  type Object is tagged;
end Department;

with Employee;
package Department is
  type Object is tagged private;
  procedure Choose_Manager (For_Department : in out Department.Object;
      Who            : in out Employee.Object);
private
  type Object is tagged
    record
      Manager : access Employee.Object;
    end record;
end Department;
```

Implicit!
Solving the Cyclic Type Conundrum

```ada
package Department is
type Object is tagged;
end Department;

limited with Department;
package Employee is
type Object is tagged private;
procedure Assign_Employee (Who            : in out Employee.Object;
                           To_Department  : in out Department.Object);
private
  type Object is tagged
  record
    Assigned_To : access Department.Object;
  end record;
end Employee;

with Employee:
package Department is
type Object is tagged private;
procedure Choose_Manager (For_Department : in out Department.Object;
                         Who            : in out Employee.Object);
private
  type Object is tagged
  record
    Manager : access Employee.Object;
  end record;
end Department;
```
Language Design Principles

- A hard problem to solve in Ada!
  - Seven different proposals studied by the ARG

- Avoid “ripple effect”
  - Adding or removing a with clause from a unit changes the legality of some other unit that depends on it
  - Maintenance headache and incomprehensible errors
  - Implementation difficulties

- Significant because the addition or removal of a with clause may create or remove cycles
  - The rules avoid ripple effects, but the user can ignore the details
Language Design Principles and Restrictions

- Detect errors early
  - References to types declared in limited views checked at compile time

- Limited view must be constructible from purely syntactic information
  - Constructs that require name resolution are not part of the limited view
  - Package renamings and instantiations
  - Tagged-ness may be determined syntactically

- Limited with clauses used to resolve circularities, not to restrict visibility
  - Limited with clause not allowed if there is already a normal with clause
  - Limited with clause not allowed on a body
  - Limited with clause not allowed with use clauses
Incomplete Types and Dereferencing

- Access types declared using the limited view are access-to-incomplete
  - Would not be very useful because of the restrictions on incomplete types

- Become access-to-complete in the presence of a nonlimited with clause

```ada
limited with Department;
package Employee is
  ...;
private
  type Object is tagged record
    Assigned_To : access Department.Object;
  end record;
end Employee;

with Department;
package body Employee is
  An_Employee    : Employee.Object   := ...;
  Her_Department : Department.Object := An_Employee.Department.all;
  ...;
end Employees;
```

This with clause ...

... makes this dereference legal
Overview

- Multi-package type structures
- Access to private units in private parts
  - Instantiating generics with private types
  - Partial parameter lists for formal instantiations
- Making limited types useful
Visibility and Program Structure (again)

- Huge changes with respect to visibility in Ada 95
- Introduction of hierarchical library units
  - Public and private children
- ... but another problem has remained...
Access to Private Units in Private Parts

- Private child packages allow decomposition and hiding of the implementation details
  - Not visible to the outside world
- Only private packages and bodies can reference a private child
- Often convenient for public packages to use implementation details without making them visible
- Impossible to use a private unit in declarations appearing in the private part of a public package
Solution: Private With Clause

- Private with clause gives visibility to a unit, but only at the beginning of the private part

```ada
private package App.Secret_Details is
  type Inner is …;

  … -- Various operations on Inner, etc.
end App.Secret_Details;

private with App.Secret_Details;
package App.User_View is
  type Outer is private;

  … -- Various operations on Outer visible to the user
     -- Type Inner may not be used here.
private
type Outer is
  record
    Secret : Secret_Details.Inner;
  end record;
end App.User_View;
```
Overview

- Multi-package type structures
- Access to private units in private parts
- Instantiating generics with private types
- Partial parameter lists for formal instantiations
- Making limited types useful
Instantiating Generics with Private Types

- A private type may be used as a component of an array or a record
  - Even before the type is complete

- It may not be used to instantiate a generic
  - Not before the type is complete
  - Problematic for using fancy containers

```ada
type Window is tagged private;
type Windows is array (Positive range <>) of Window; -- Fine.

type Window is tagged private;
package Vectors_Of_Windows is
```
Solution: Partial Package Instantiations

- Package instantiations may (but need not) come in two parts
- Partial instantiation may use private types
  - Exports entities that “look private”
  - Cannot be used to create objects, compute expressions, etc.
- Full instantiation given later after the type has been completed

```ada
type Window is tagged private;

package Vectors_Of_Windows is
  new Ada.Containers.Vectors (..., Window, ...) with private;
  ...
private
  type Window is tagged record ... end record;

package Vectors_Of_Windows is
  new Ada.Containers.Vectors (..., Window, ...);
```
Overview

- Multi-package type structures
- Access to private units in private parts
- Instantiating generics with private types
- Partial parameter lists for formal instantiations
- Making limited types useful
Ada 95 introduced formal packages as parameters of generics
- Encapsulate several generic formal parameters
- Reduced the need for long, hard-to-maintain, parameter lists

Each formal package may put requirements on its instantiation parameters
- Either “anything goes”: <> as actual parameter part
- Or “specify all the details”: explicit names and values given for all the parameters

No way to impose “partial requirements”
Solution: Partial Parameter Lists

- Ada.Containers.Vectors
  - Index_Type, Element_Type, “=” on Element_Type

- Ada.Containers.Doubly_Linked_Lists
  - Element_Type, “=” on Element_Type

- Generic function to convert a vector into a list
  - Vector and list must agree on the Element_Type and the “=” operator
  - Anything goes for Index_Type

```ada
generic
  with package Lists is new Ada.Containers.Doubly_Linked_Lists (<>);
  with package Vectors is new Ada.Containers.Vectors
    (Index_Type => <>,
     Element_Type => Lists.Element_Type,
     "=" => Lists."=");

function Convert (V : Vectors.Vector) return Lists.List;
```
Overview

- Multi-package type structures
- Access to private units in private parts
- Instantiating generics with private types
- Partial parameter lists for formal instantiations
- Making limited types useful
Making Limited Types Useful

- Limited types prevent copying of values
  - Have limitations unrelated to copying

- Aggregates not available: no full coverage checking

- Functions cannot be used to construct values of limited types
  - Can only return existing global objects: not too useful
  - Mysterious “return by reference” mechanism

- Limited types are unnecessarily hard-to-use
  - Restrictions do not improve safety
  - Types often made nonlimited to avoid running into difficulties

- Lift unnecessary restrictions while preserving safety
  - In particular, prevent copying of values
Solution: Aggregates for Limited Types

- Aggregates only allowed for initialization, not general assignment
  - Must be built in place

- New syntax for components for which no value can be written
  - Tasks, protected objects
  - Also causes default initialization if a default value was given in the declaration

```ada
protected type Semaphore is ...;

type Object is limited
record
  Guard    : Semaphore;
  Value    : Float;
  Finished : Boolean := False;
end record;

type Ptr is access Object;

X : Ptr := new Object'(Guard => <>, -- A new Semaphore.
                           Value   => 0.0,
                           Finished => <> -- Gets False.
                          );
```
Solution: Functions Returning Limited Types

- Again, only allowed for initialization
- New form of return statement to build an object directly in its final resting place
  - No copying of the result of the function

```ada
function Random_Object return Object is
  use Ada.Numerics.Float_Random;
  Gen : Generator;
begin
  Reset (Gen);
  return New_Object : Object do
  New_Object.Value := Random (Gen);
  New_Object.Finished := New_Object.Value > 0.5;
  end return;
end Random_Object;
```
Real-Time Improvements

Alan Burns, University of York
Overview

- Ravenscar
- Support for control over execution time
- Timing Events
- Dynamic ceiling priorities for Protected Objects
- Support for additional scheduling/dispatching
The Ravenscar Profile

- A subset of the Ada tasking model
  - Silent on the sequential part of the language

- Restrictions designed to meet the real-time community requirements for
  - Determinism
  - Schedulability analysis
  - Memory-boundedness
  - Execution efficiency and small footprint
  - Suitability for certification
The Ravenscar Profile

- The Ravenscar Profile is a powerful catalyst for the promotion of simple and effective language-level concurrency
  - Crucial to critical applications
  - One end of the road to greater expressive power
Ravenscar

- Profile uses a set of Restrictions
  - Max_Task_Entries => 0
  - Max_Protected_Entries => 1
  - No_Abort_Statements
  - No_Asynchronous_Control
  - No_Dynamic_Priorities
  - No_Implicit_Heap_Allocations
  - No_Task_Allocators
  - No_Task_Hierarchy
Ravenscar

- New restriction identifiers
  - Max_EntryQueueLength => 1
  - No_Calendar
  - No_Dynamic_Attachment
  - No_Local_Protected_Types
  - No_Protected_Type_Allocators
  - No_Relative_Delay
  - No_Requeue_Statements
  - No_Select_Statements
  - No_Task_Attributes_Package
  - No_Task_Termination
  - Simple_Barriers
Ravenscar

- New pragma:
  - pragma Detect_Blocking

- Dispatching
  - FIFO_Within_Priorities
  - Ceiling_Locking

- New pragma for defining a profile:
  - pragma Profile();
The Ravenscar Profile

- The profile corresponds to:

```ada
pragma Task_Dispatching_Policy (FIFO_Within_Priorities);
pragma Locking_Policy (Ceiling_Locking);
pragma Detect_Blocking;
pragma Restrictions (Max_Entry_Queue_Length => 1,
                      Max_Protected_Entries => 1,
                      Max_Task_Entries => 0,
                      No_Abort_Statements,
                      No_Async_Control,
                      No_Calendar,
                      No_Dynamic_ATTACHMENT,
                      No_Dynamic_Priorities,
                      No_Implicit_Heap_Allocations,
                      No_Local_Protected_Objects,
                      No_Protected_Type_Allocators,
                      No_Relative_Delay,
                      No_Requeue_Statements,
                      No_Select_Statements,
                      No_Task_Allocators,
                      No_Task_Attributes_Package,
                      No_Task_Hierarchy,
                      No_Task_Termination,
                      Simple_Barriers);
```
Examples of Use

task type Cyclic (Pri : System.Priority;
   Cycle_Time : Positive) is
   pragma Priority (Pri);
end Cyclic;

task body Cyclic is
   Next_Period : Ada.Real_Time.Time;
   Period      : constant Ada.Real_Time.Time_Span :=
                    Ada.Real_Time.Microseconds (Cycle_Time);
   -- Other declarations.
begin
   -- Initialization code.
   Next_Period := Ada.Real_Time.Clock + Period;
   loop -- Wait one whole period before executing.
      delay until Next_Period;
      -- Non-suspending periodic response code.
      -- May include calls to protected procedures.
      Next_Period := Next_Period + Period;
   end loop;
end Cyclic;

-- 2 task objects of this type.
A_Cyclic_Task : Cyclic (20,200);
Another_Cyclic_Task : Cyclic (15,100);
Examples of Use

-- A suspension object SO is declared in a visible library unit
-- and is set to True in another (releasing) task.

task type Sporadic (Pri : System.Priority) is
  pragma Priority (Pri);
end Sporadic;

task body Sporadic is
  -- Declarations.
begin
  -- Initialization code.
  loop
    Ada.Synchronous_Task_Control.Suspend_Until_True (SO);
    -- Non-suspending sporadic response code.
    end loop;
end Sporadic;

An_Event_Triggered_Task : Sporadic (17);
Examples of Use

protected type Event (Ceiling : System.Priority) is
  entry Wait  (D :  out Data);
  procedure Signal (D :  in  Data);
private
  -- Ceiling priority defined for each object.
  pragma Priority (Ceiling);
  Current   : Data; -- Event data declaration.
  Signalled : Boolean := False;
end Event;

protected body Event is
  entry Wait (D :  out Data) when Signalled is
  begin
    D := Current;
    Signalled := False;
  end Wait;
  procedure Signal (D :  in  Data) is
  begin
    Current  := D;
    Signalled := True;
  end Signal;
end Event;
Examples of Use

Event_Object : Event (15);

task Event_Handler is
  pragma Priority (14); -- Must be not greater than 15.
end Event_Handler;

task body Event_Handler is
  -- Declarations, including D of type Data.
begin
  -- Initialization code.
  loop
    Event_Object.Wait(D);
    -- Non-suspending event handling code.
  end loop;
end Event_Handler;
Execution Time Support

- Monitor the task execution time
- Fire an event when a task execution time reaches a specified value
- Allocate and support budgets for groups of tasks
Monitoring Task Execution Time

- Every task has an execution time clock
- Clock starts sometime between creation and start of activation
- Clock counts up whenever the task executes
- Accuracy, metrics and implementation requirements defined
Monitoring Task Execution Time (cont’d)

```ada
with Ada.Task_Identification;
with Ada.Real_Time; use Ada.Real_Time;
package Ada.Execution_Time is

  type CPU_Time is private;
  CPU_Time_First : constant CPU_Time;
  CPU_Time_Last  : constant CPU_Time;
  CPU_Time_Unit  : constant :=
                      implementation-defined-real-number;
  CPU_Tick : constant Time_Span;

  function Clock
    return CPU_Time;

    -- Subprograms for + etc, < etc, Split and Time_Of.

private
  ... -- Not specified by the language.
end Ada.Execution_Time;
```
Triggering

- In fault tolerance and other high integrity applications there is a need to catch task overruns

- For some algorithms a fixed time is allocated to a task for some iterative process

- Basic model is to define:
  - A timer that is enabled, and
  - A handler that is called (by the run-time) when a task’s execution time clock become equal to some defined value

- The handler is a not null access to protected procedure
Triggering (cont’d)

package Ada.Execution_Time.Timers is

  type Timer (T : access Ada.Task_Identification.Task_ID) is limited private;
  type Handler is not null access protected
  procedure (TM : in out Timer);

  Min_Handler_Ceiling : constant System.Any_Priority := <Implementation Defined>;

  procedure Arm (TM : in out Timer;
                 Interval : Time_Span; H : Handler);
  procedure Arm (TM : in out Timer;
                 Abs_Time : CPU_Time; H : Handler);
  procedure Disarm(TM : in out Timer);

  function Timer_Has_Expired(TM : Timer) return Boolean;
  function Time_Remaining(TM : Timer) return Time_Span;

  Timer_Error : exception;
  Timer_Resource_Error : exception;

end Ada.Execution_Time.Timers; -- There is a private part.
Budget Scheduling

- A number of schemes, including those that use servers allow a group of tasks to share a budget
- The budget is usually replenished periodically
- The scheme support fires a handler when budget goes to zero
  - The tasks are not prevented from executing
  - But this can be programmed
  - or priorities change to background, or whatever…
package Ada.Execution_Time.Group_Budgets is
  type Group_Budget is limited private;

  type Handler is not null access protected
    procedure (GB : in out Group_Budget);

  type Task_Array is array (Natural range <>) of
    Ada.Task_Identification.Task_ID;

  Min_Handler_Ceiling : constant System.Any_Priority :=
    <Implementation Defined>;

  procedure Add_Task(GB : in out Group_Budget;
                     T : Ada.Task_Identification.Task_ID);

  procedure Remove_Task(GB : in out Group_Budget;
                        T : Ada.Task_Identification.Task_ID);

  function Is_Member(GB : Group_Budget;
                     T : Ada.Task_Identification.Task_ID) return Boolean;

  function Is_A_Group_Member( T : Ada.Task_Identification.Task_ID) return Boolean;

  function Members(GB : Group_Budget) return Task_Array;

  ...
Budget Scheduling (cont’d)

... procedure Replenish (GB: in out Group_Budget; To : Time_Span);
procedure Add(GB: in out Group_Budget; Interval : Time_Span);
function Budget_Has_Expired(GB: Group_Budget) return Boolean;
function Budget_Remaining(GB: Group_Budget) return Time_Span;

procedure Set_Handler(GB: in out Group_Budget; H : Handler);
function Current_Handler(GB: Group_Budget) return Handler;
procedure Cancel_Handler(GB: in out Group_Budget;
    Cancelled : out Boolean);

Group_Budget_Error : exception;
private
    -- Not specified by the language.
end Ada.Execution_Time.Group_Budgets;
Timing Events

- A means of defining code that is executed at a future point in time
- Does not need a task
- Similar in notion to interrupt handing (time itself generates the interrupt)
- Again a handler is used
package Ada.Real_Time.Timing_Events is
    type Timing_Event is limited private;
    type Timing_Event_Handler is access protected
        procedure (Event : in out Timing_Event);
    procedure Set_Handler(Event : in out Timing_Event;
        At_Time : Time; Handler: Timing_Event_Handler);
    procedure Set_Handler(Event : in out Timing_Event;
        In_Time: Time_Span; Handler: Timing_Event_Handler);
    function Is_Handler_Set(Event : Timing_Event) return Boolean;
    function Current_Handler(Event : Timing_Event) return Timing_Event_Handler;
    procedure Cancel_Handler(Event : in out Timing_Event;
        Cancelled : out Boolean);
    function Time_Of_Event(Event : Timing_Event) return Time;
private
    ... -- Not specified by the language.
end Ada.Real_Time.Timing_Events;
Example of Usage

```ada
protected Watchdog is
pragma Interrupt_Priority (Interrupt_Priority'Last);
entry Alarm_Control;
   -- Called by alarm handling task.
procedure Timer(Event : in out Timing_Event);
   -- Timer event code.
procedure Call_in;
   -- Called by application code every 50ms if alive.
private
   Alarm : Boolean := False;
end Watchdog;

Fifty_Mil_Event : Timing_Event;
TS : Time_Span := Milliseconds(50);

Set_Handler(Fifty_Mil_Event, TS, Watchdog.Timer'Access);
```
Example of Usage (cont’d)

protected body Watchdog is
  entry Alarm_Control when Alarm is
  begin
    Alarm := False;
  end Alarm_Control;

procedure Timer(Event : in out Timing_Event) is
begin
  Alarm := True;
end Timer;

procedure Call_in is
begin
  Set_Handler(Fifty_Mil_Event, TS, Watchdog.Timer'access);
  -- Note, this call to Set_Handler cancels the previous call.
end Call_in;
end Watchdog;
Dynamic Ceilings

- A new attribute for any protected object: ‘Priority
- This attribute can be read and assigned to within the body of a PO (only)
- The effect of any change to the ceiling of the PO takes effect at the end of the current protected action
Ada Rapporteur Group

Scheduling and Dispatching

- Ada provides a complete and well defined set of language primitives for fixed priority scheduling
- But fixed priority scheduling is not the only scheme of interest
- The amendment to Ada allows the language to define other schemes
- The authority of the language definition is needed to sanction there schemes
Dispatching Policies

- **Fixed Priority**
  - Still the main dispatching policy

- **Some changes to Annex D needed to allow the following:**
  - Non-preemptive
    - `Non_Preemption_Within_Priority`
  - Round Robin
  - EDF
  - Mixed policies within a partition
package Ada.Dispatching is
  pragma Pure(Dispatching);
  Dispatching_Policy_Error : exception;
end Ada.Dispatching;
Round Robin

- A common policy in non-real-time systems and in some real-time schemes requiring a level of fairness
- Require a simple scheme with the usual semantics
- If the defined quantum is exhausted during the execution of a protected object then the task involved continues executing until it leaves the protected object
- A support package is provided
Round Robin (cont’d)

with System;
with Ada.Real_Time;
package Ada.Dispatching.Round_Robin_Dispatching is
   Default_Quantum : constant Ada.Real_Time.Time_Span :=
      <implementation-defined>;
   procedure Set_Quantum(Pri : System.Priority;
      Quantum : Ada.Real_Time.Time_Span);
   procedure Set_Quantum(Low,High : System.Priority;
      Quantum : Ada.Real_Time.Time_Span);
   function Actual_Quantum
      (Pri : System.Priority) return
      Ada.Real_Time.Time_Span;
   function Is_Round_Robin (Pri : System.Priority) return
      Boolean;
end Ada.Dispatching.Round_Robin_Dispatching;
Deadlines and EDF Scheduling

- The **deadline** is the most significant notion in real-time systems
- EDF – Earliest Deadline First is the scheduling policy of choice in many domains
- It makes better use of processing resources
- EDF or FP?
  - a long and detailed debate
  - but in reality both are needed
Support for Deadlines

- Introduction of a new library package
- **Relative deadline** means relative to task’s release
  - complete talk in 30 minutes
- **Absolute deadline** means point on time line
  - complete talk by 12.30
- Usually **deadline** means absolute deadline
Support for Deadlines (cont’d)

```ada
with Ada.Task_Identification;
use Ada.Task_Identification;
with Ada.Real_Time;
package Ada.Dispatching.EDF_Dispatching is
    subtype Deadline is Ada.Real_Time.Time;
    procedure Set_Deadline(
        D : Deadline;
        T : Task_ID := Current_Task);
    function Get_Deadline(
        T : Task_ID := Current_Task)
        return Deadline;
    procedure Delay_Until_And_Set_Deadline(
        Delay_Until_Time : Ada.Real_Time.Time;
        TS : Ada.Real_Time.Time_Span);
end Ada.Dispatching.EDF_Dispatching;
```
Catching a Deadline Overrun

loop
  select
    delay until Deadlines.Get_Deadline;
    -- Deal with deadline overrun.
  then abort
    -- Code.
  end select;
  -- Set next release condition
  -- and next absolute deadline.
end loop;
EDF Scheduling

- Need to define EDF ordered ready queues
- Need to support preemption-level locking for effective use of protected objects
  - Ideally uses existing PO locking
  - Ideally can be used with fixed priority scheduling
Baker’s Protocol

- Under Fixed Priority scheduling, priority is used for:
  - Dispatching
  - Controlled access to resources eg Pos

- Under Baker’s protocol
  - Dispatching is controlled by absolute deadline
  - Preemption levels used for resources
Baker’s Protocol

- Basic algorithm
  - A newly released task can preempt the currently executing task iff:
    - Its deadline is earlier
    - Its preemption-level is greater than that of the highest locked resource
Bounding Blocking

- If preemption levels are assigned according to relative deadline then we can have:
  - Deadlock free execution
  - Maximum of one block per invocation

- Hence same properties as priority ceiling protocol for FP systems
  - i.e., Ada’s existing model for POs
Dispatching Rules for EDF

- Use a task’s base priority to represent preemption level
- Assigned PO ceiling priorities (preemption levels) in the usual way
  - execution within a PO is at ceiling level
- Order ready queues by absolute deadline
Which Queue to Join?

- Define a ready queue at priority level $p$ as being **busy** if a task has locked a PO with ceiling $p$ – denote this task as $T(p)$

- A newly released task $S$ is added to highest priority busy ready queue $p$ such that deadline of $S$ is earlier than $T(p)$ and base priority of $S$ is greater than $p$

- **If no $p$ exist put $S$ on Priority’ First**
Properties

- Task S is always placed on a priority level below that of the ceiling priority of any PO it uses
- Implements Baker’s protocol
- Splitting the priority range into FP and EDF allows both to work together
Example

- Following slide has one cyclic task of a simple system of 5 tasks with preemption levels 1..5

- Dispatched by:

  \texttt{pragma Task_Dispatching_Policy (FIFO\_Within\_Priorities);}
Example (cont’d)

protected X is -- one of 3 POs  
   pragma Priority(5);  
   -- Definitions of subprograms.
private  
   -- Definition of internal data.
end X;

task A is  
   pragma Priority(5);  
   -- Period and  
   end A;  
   -- relative deadline equal to 10ms.

task body A is  
begin  
   loop  
      -- Code, including call(s) to X.  
      Next_Release := Next_Release +  
         Ada.Real_Time.Milliseconds(10);  
      delay until Next_Release;  
   end loop;  
end A;
task A is
  pragma Priority(5);
  pragma Relative_Deadline(10);
end A;

task body A is
begin
  loop
    -- Code, including call(s) to X.
    Next_Release := Next_Release +
      Ada.Real_Time.Milliseconds(10);
    Deadlines.Set_Deadline(Next_Release +
      Ada.Real_Time.Milliseconds(10));
    delay until Next_Release;
  end loop;
end A;

----------
pragma Task_Dispatching_Policy
  (EDF_Across_Priorities);
Example (cont’d)

```
  task body A is
  begin
      loop
          -- code, including call(s) to X
      Next_Release := Next_Release +
                      Ada.Real_Time.Milliseconds(10);
      Deadline.Delay_Until_And_Set_Deadline
          (Next_Release,
           Ada.Real_Time.Milliseconds(10));
  end A;
```
Mixed Dispatching

- Ada now allows different dispatching policies to be used together in a controlled and predictable way.

- Protected object can be used to communicate across policies.

```ada
pragma Priority_Specific_Dispatching(
    policy_identifier,
    first_priority_expression,
    last_priority_expression);
```
Splitting the Priority Range

`pragma Priority_Specific_Dispatching`  
    (Round_Robin_Within_Priority, 1, 1);

`pragma Priority_Specific_Dispatching`  
    (EDF_Across_Priorities, 2, 10);

`pragma Priority_Specific_Dispatching`  
    (FIFO_Within_Priority, 11, 24);
Conclusions

- The amendment to Ada has significantly extended the facilities available for programming real-time systems
  - Ravenscar, execution time control, timing events, dispatching

- The requirements for these changes have come from the series of International Real-Time Ada Workshops

- Ada is now considerably more expressive in this area than any other programming language
Overview

- Vectors and matrices (13813++)
- Directories
- Environment variables
- More string subprograms
- Wider and wider
- Containers
- Time zones and leap seconds
Vectors and Matrices

- Incorporates missing stuff from ISO/IEC 13813
- Generic packages
  - Ada.Numerics.Generic_Real_Arrays
  - Ada.Numerics.Generic_Complex_Arrays
- These contain various arithmetic operations +, -, * acting on vectors and matrices
- Also Transpose, Conjugate, etc. all as in 13813
- Plus
  - Linear equations
  - Inverse, determinant, eigenvalues and vectors
Simple Arithmetic

- Given vectors $x$, $y$, $z$ and square matrix $A$
  To perform the mathematical computation $y = Ax + z$

- We simply write

```ada
X, Y, Z : Real_Vector(1 .. N); -- Types from
A : Real_Matrix (1 .. N, 1 .. N); -- Generic_Real_Arrays.
...
Y := A * X + Z; -- Ops from ditto.
```

- All works perfectly – designed by Numerics Rapporteur Group in the previous century
Solve Linear Equations

- Again if \( y = Ax + z \), to compute \( x \) given \( A, y \) and \( z \),

  That is \( x = A^{-1}(y - z) \)

- We write

\[
X := \text{Solve}(A, Y - Z);
\]
Also

- To invert a matrix

\[ B := \text{Inverse}(A); \]

- To compute determinant

\[ \text{Det} := \text{Determinant}(A); \]

- To find eigenvalues

\[ \text{Values} := \text{Eigenvalues}(A); \quad -- \text{Symmetric/Hermitian} \]
Overall Goals

- To incorporate the features of 13813
- To provide a foundation for bindings to libraries such as the BLAS (Basic Linear Algebra System)
- To make simple, frequently used, linear algebra operations immediately available without fuss
Directories

- package Ada.Directories provides
  - Directory and file operations
  - File and directory name operations
  - File and directory queries
  - Directory searching
  - Operations on directory entries

- Enables one to mess about with file names, extensions and so on

- They tell me it is jolly good for Unix and Windows
Environment Variables

- package Ada.Environment_Variables
- Enables one to peek and poke at OS variables
More String Subprograms

- Problems with 95
  - Conversions between Bounded_String and String and between Unbounded_String and String are required rather a lot
    - Ugly & inefficient
  - Thus searching part of a bounded or unbounded string requires converting to String first
- So further subprograms added
Further Index Subprograms

- With additional parameter From such as

  ```ada
  function Index (Source: in Bounded_String;
                  Pattern: in String;
                  From: in Positive;
                  Going: in Direction := Forward;
                  Mapping: in Maps.Character_Mapping := ...)
  return Natural;
  ```

- Also with Source of types String and Unbounded_String

- And Index_Non_Blank
More

- Function and procedure Bounded_Slice and Unbounded_Slice
  - Avoid conversions to type String

- A new package Ada.Text_IO.Unbounded_IO
  - Also avoids conversions to String
  - (not for Bounded_IO because of generic complexity)

- And functions Get_Line for Ada.Text_IO
  - The existing procedures are awkward
More Identifier Freedom

- Ada 83 – identifiers in 7-bit ASCII
  boy, devil, goat

- Ada 95 – identifiers in 8-bit Latin-1
  garçon, dæmon, chèvre

- Ada 2005 – identifiers in 16-bit BMP++
  мальчик, демон, коза

Сталин : access Pig renames Napoleon;
Πεγασος : Horse;
Wider and Wider

- Ada 83 has
  - Character and String

- Ada 95 also has
  - Wide_Character and Wide_String

- Ada 2005 also also has
  - Wide_Wide_Character and Wide_Wide_String
Containers

- This should be a whole lecture in itself

- A package Ada.Containers plus lots of children
  - Ada.Containers.Vectors
  - Ada.Containers.Doubly_Linked_Lists
  - Ada.Containers.Hashed_Maps
  - Ada.Containers.Ordered_Sets
    - also indefinite versions of the above
  - Ada.Containers.Generic_Array_Sort
    - and constrained version
Vectors & Lists

- Uniform approach, many routines common, thus
- Elements can be referred to
  - By cursor
- Insert, Append, Prepend, Delete, etc.
- Various searching, sorting and iterating procedures, e.g.,

```ada
procedure Iterate
  (Container : in Vector/List;
   Process : not null
             access procedure (Position : in Cursor));
```

- Note anonymous access to subprogram parameter
Maps & Sets

- Uniform approach, many routines common, thus

- Elements can be referred to
  - By cursor

- Insert, Delete etc (not Append, Prepend)

- Also iterating procedure (not searching, sorting)

```ada
procedure Iterate
(Container: in Maps/Sets;
 Process: not null access procedure (Position: in Cursor));
```
General Array Sorting

generic
    type Index_Type is (<>);
    type Element_Type is private;
    type Array_Type is array (Index_Type range <>)
        of Element_Type;

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

procedure Ada.Containers.Generic_Array_Sort
    (Container: in out Array_Type);
Overall Goals

- Provide the most commonly required data structure routines
- Use uniform approach where possible so that conversion is feasible
- Make them reliable
  - thou shalt not corrupt thy container
More Calendar

- Three children of calendar
  Ada.Calendar.Time_Zones
  Ada.Calendar.Arithmetic
  Ada.Calendar.Formatting

- Why not just one child package?
  - To be honest -
  - No sensible name - Ada.Calendar.MoreStuff not appropriate

- Main goals
  - Deal with time zones and leap seconds
But

- Everyone will appreciate

```ada
type Day_Name is (Monday, Tuesday, Wednesday,
                  Thursday, Friday, Saturday, Sunday);
function Day_Of_Week(Date: Time) return Day_Name;
```

- Also, Year_Number is extended

```ada
subtype Year_Number is Integer range 1901 .. 2399;
```

- Another 300 years. Long live Ada!!
The End of Me

- Gosh it must be nearly time for lunch
- But first an important message from Tucker on safety
Safety in Ada 2005

S. Tucker Taft, SofCheck, Inc.
Ada 2005 Safety-Related Amendments

- Syntax to prevent unintentional overriding or non-overriding of primitive operations
  - Catch spelling errors, parameter profile mismatches, maintenance confusion

- Standardized Assert pragma
  - Assertion_Policy pragma determines how Assert is handled by implementation (Check, Ignore, …)

- Standardized Unsuppress pragma

- Standardized No_Return pragma
  - Identifies routines guaranteed to never return to point of call
Ada 2005 Safety-Related Amendments (cont’d)

- Availability of “not null” and “access constant” qualifiers for access parameters
- Standardized high-integrity “Ravenscar” profile
- Handlers for unexpected task termination
Control of Overriding

- Can specify that an operation is overriding an inherited primitive operation
- Can specify that an operation is *not* overriding any inherited primitive
- Can specify nothing, which is the current situation, where overriding is allowed, but not required

```ada
type File_Stream is new Root_Stream_Type with private;

overriding
procedure Read(Stream : in out File_Stream;
    Item : out Stream_Element_Array;
    Last : out Stream_Element_Offset);

not overriding
procedure Read_All(Stream : in out File_Stream;
    Content : out Unbounded_String);
```
Control of Overriding (cont’d)

- Specifying “overriding” protects against spelling errors, wrong order or types of parameters, etc.
- Specifying “not overriding” protects against unintentional overriding
  - Can be particularly important in generics
Control of Overriding (cont’d)

- For a generic, “not overriding,” if specified, must be true both:
  - When the generic (template) is compiled
  - When the generic is instantiated

```ada
generic
  type Node is new Base with private;
package Linked_Lists is
  type List_Element is new Base with private;

not overriding
function Next(LE : access constant List_Element) return access List_Element’Class;

not overriding
procedure Set_Next(LE : access List_Element;
  Next : access List_Element’Class);
```
Safety-Related Pragmas

- **pragma** Assert(X /= 0, “cot(0) not defined”);
  - Already supported by most Ada 95 compilers
  - Now can be used portably

- **pragma** Assertion_Policy(Check);
  - Standardized way to control enforcement of Assert pragmas
  - “Check” and “Ignore” are language-defined policies
    - Implementation may define additional policies
Safety-Related Pragmas

- **pragma** Unsuppress(Overflow_Check);
  - Ensure that algorithm that depends on constraint check will work properly, even in presence of Suppress pragmas

- **pragma** No_Return(Fatal_Error);
  - Identify procedure that never returns to point of call
  - Improves static analysis possible for compiler or other tools
  - Raises Program_Error if procedure attempts to return
Safety Is Our Most Important Product

- Ada is the premier language for safety critical software
- Ada’s safety features are critical to making Ada such a high-productivity language in all domains
- Amendments to Ada carefully designed so as to not open any new safety holes
- Several amendments provide even more safety, more opportunities for catching mistakes at compile-time
It Really is Time for Lunch