Overview

About This Course

Styles

- *This* is a definition
- `this/is/a.path`
- `code is highlighted`
- `commands are emphasised --like-this`

A Little History

The Name

- First called DoD-1
- Augusta Ada Byron, "first programmer"
  - Lord Byron's daughter
  - Planned to calculate **Bernouilli's numbers**
  - **First** computer program
  - On **Babbage's Analytical Engine**
- Writing **ADA** is like writing **CPLUSPLUS**
- International Standards Organization standard
  - Updated about every 10 years

Ada Evolution Highlights

Ada 83
- Abstract Data Types
- Modules
- Concurrency
- Generics
- Exceptions

Ada 95
- OOP
- Efficient synchronization
- Better Access Types
- Child Packages
- Annexes

Ada 2005
- Multiple Inheritance
- Containers
- Better Limited Types
- More Real-Time
- Ravenscar

Ada 2012
- Contracts
- Iterators
- Flexible Expressions
- More containers
- Multi-processor Support
- More Real-Time

Ada 2022
- `'Image` for all types
- Target name symbol
- Support for C varidics
- Declare expression
- Simplified `renames`

Big Picture

Language Structure (Ada95 and Onward)

- **Required** *Core* implementation
  - Reference Manual (RM) sections 1 $\rightarrow$ 13
  - Predefined Language Environment (Annex A)
  - Interface to Other Languages (Annex B)
  - Obsolescent Features (Annex J)
- Optional *Specialized Needs Annexes*
  - No additional syntax
  - Systems Programming (C)
  - Real-Time Systems (D)
  - Distributed Systems (E)
  - Information Systems (F)
  - Numerics (G)
  - High-Integrity Systems (H)

Core Language Content

- Ada is a **compiled**, **multi-paradigm** language
- With a **static** and **strong** type model

- Language-defined types, including string
- User-defined types
- Overloading procedures and functions
- Compile-time visibility control
- Abstract Data Types (ADT)
- Exceptions
- Generic units
- Dynamic memory management
- Low-level programming
- Object-Oriented Programming (OOP)
- Concurrent programming
- Contract-Based Programming

Ada Type Model

- **Static** Typing
  - Object type **cannot change**
  - ... but run-time polymorphism available (OOP)
- **Strong** Typing
  - **Compiler-enforced** operations and values
  - **Explicit** conversions for "related" types
  - **Unchecked** conversions possible
- Predefined types
- Application-specific types
  - User-defined
  - Checked at compilation and run-time

Strongly-Typed vs Weakly-Typed Languages

- Weakly-typed:
    - Conversions are **unchecked**
    - Type errors are easy

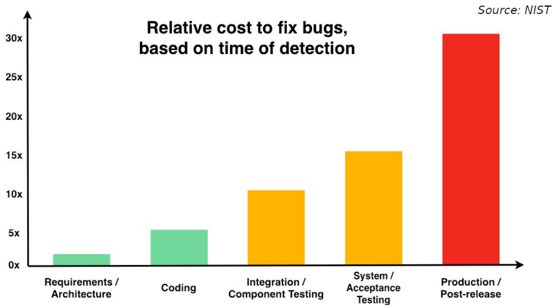
```
typedef enum {north, south, east, west} direction;
typedef enum {sun, mon, tue, wed, thu, fri, sat} days;
direction heading = north;

heading = 1 + 3 * south/sun;// what?
```

- Strongly-typed:
    - Conversions are **checked**
    - Type errors are hard

```
type Directions is (North, South, East, West);
type Days is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
Heading : Directions := North;
...
Heading := 1 + 3 * South/Sun; --  Compile Error
```

The Type Model Saves Money

- Shifts fixes and costs to **early phases**
- **Cheaper**
  - Cost of an error *during a flight*?



Type Model Run-Time Costs

- Checks at compilation **and** run-time
- **Same performance** for identical programs
  - Run-time type checks can be disabled
  - Compile-time check is *free*

C

```c
int X;
int Y; // range 1 .. 10
...
if (X > 0 && X < 11)
  Y = X;
else
  // signal a failure
```

Ada

```ada
X : Integer;
Y, Z : Integer range 1 .. 10;
...
Y := X;
Z := Y; -- no check required
```

Subprograms

- Syntax differs between *values* and *actions*
- **function** for a *value*

```
function Is_Leaf (T : Tree) return Boolean
```

- **procedure** for an *action*

```
procedure Split (T     : in out Tree;
                 Left  : out Tree;
                 Right : out Tree)
```

- Specification $\neq$ Implementation

```
function Is_Leaf (T : Tree) return Boolean;
function Is_Leaf (T : Tree) return Boolean is
begin
...
end Is_Leaf;
```

Dynamic Memory Management

- Raw pointers are error-prone
- Ada **access types** abstract facility
  - Static memory
  - Allocated objects
  - Subprograms
- Accesses are **checked**
  - Unless unchecked mode is used
- Supports user-defined storage managers
  - Storage **pools**

Packages

- Grouping of related entities
  - Subsystems like *Fire Control* and *Navigation*
  - Common processing like *HMI* and *Operating System*
- Separation of concerns
  - Definition $\neq$ usage
  - Single definition by **designer**
  - Multiple use by **users**
- Information hiding
  - Compiler-enforced **visibility**
  - Powerful **privacy** system

Package Structure

- Declaration view
    - **Can** be referenced by user code
    - Exported types, variables...
- Private view
    - **Cannot** be referenced by user code
    - Exported **representations**
- Implementation view
    - Not exported

Abstract Data Types (ADT)

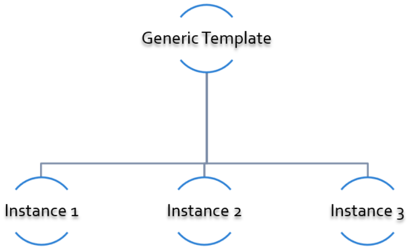
- **Variables** of the **type** encapsulate the **state**
- Classic definition of an ADT
  - Set of **values**
  - Set of **operations**
  - **Hidden** compile-time **representation**
- Compiler-enforced
  - Check of values and operation
  - Easy for a computer
  - Developer can focus on **earlier** phase: requirements

Exceptions

- Dealing with **errors**, **unexpected** events
- Separate error-handling code from logic
- Some flexibility
    - Re-raising
    - Custom messages

Generic Units

- Code Templates
  - Subprograms
  - Packages
- Parameterization
  - Strongly typed
  - **Expressive** syntax



Object-Oriented Programming

- Extension of ADT
    - Sub-types
    - Run-time flexibility
- Inheritance
- Run-time polymorphism
- Dynamic **dispatching**
- Abstract types and subprograms
- **Interface** for multiple inheritance

Contract-Based Programming

- Pre- and post-conditions
- Formalizes specifications

```
procedure Pop (S : in out Stack) with
    Pre => not S.Empty, -- Requirement
    Post => not S.Full; -- Guarantee
```

- Type invariants

```
type Table is private with Invariant => Sorted (Table);
```

Language-Based Concurrency

- **Expressive**
  - Close to problem-space
  - Specialized constructs
  - **Explicit** interactions
- **Run-time** handling
  - Maps to OS primitives
  - Several support levels (Ravenscar...)
- **Portable**
  - Source code
  - People
  - OS & Vendors

Concurrency Mechanisms

- Task
    - **Active**
    - **Rich** API
    - OS threads
- Protected object
    - **Passive**
    - *Monitors* protected data
    - **Restricted** set of operations
    - No thread overhead
    - Very portable
- Object-Oriented
    - Synchronized interfaces
    - Protected objects inheritance

Low Level Programming

- **Representation** clauses
- Bit-level layouts
- Storage pools definition
  - With access safeties
- Foreign language integration
  - C
  - C++
  - Assembly
  - etc...
- Explicit specifications
  - Expressive
  - Efficient
  - Reasonably portable
  - Abstractions preserved

Standard Language Environment

Standardized common API

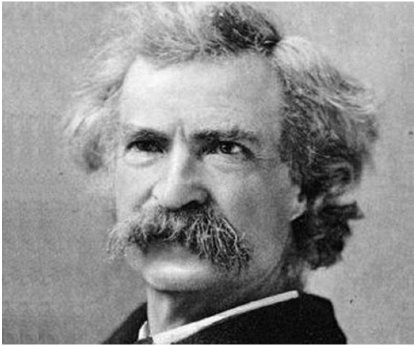
- Types
  - Integer
  - Floating-point
  - Fixed-point
  - Boolean
  - Characters, Strings, Unicode
  - etc...
- Math
  - Trigonometric
  - Complexes
- Pseudo-random number generators
- I/O
  - Text
  - Binary (direct / sequential)
  - Files
  - Streams
- Exceptions
  - Call-stack
- **Command-line** arguments
- **Environment** variables
- **Containers**
  - Vector
  - Map

Language Examination Summary

- Unique capabilities
- Three main goals
  - **Reliability**, maintainability
  - Programming as a **human** activity
  - Efficiency
- Easy-to-use
  - ...and hard to misuse
  - Very **few pitfalls** and exceptions

So Why Isn't Ada Used Everywhere?

- "... in all matters of opinion our adversaries are insane"
  - *Mark Twain*



Setup

Canonical First Program

```
1 with Ada.Text_IO;
2 -- Everyone's first program
3 procedure Say_Hello is
4 begin
5   Ada.Text_IO.Put_Line ("Hello, World!");
6 end Say_Hello;
```

- Line 1 - `with` - Package dependency
- Line 2 - `--` - Comment
- Line 3 - `Say_Hello` - Subprogram name
- Line 4 - `begin` - Begin executable code
- Line 5 - `Ada.Text_IO.Put_Line ()` - Subprogram call
- (cont) - `"Hello, World!"` - String literal (type-checked)

"Hello World" Lab - Command Line

- Use an editor to enter the program shown on the previous slide
  - Use your favorite editor or just gedit/notepad/etc.
- Save and name the file `say_hello.adb` exactly
  - In a command prompt shell, go to where the new file is located and issue the following command:
    - `gprbuild say_hello`
- In the same shell, invoke the resulting executable:
  - `say_hello` (Windows)
  - `./say_hello` (Linux/Unix)

"Hello World" Lab - GNAT Studio

- Start GNAT Studio from the command-line (`gnatstudio`) or Start Menu
- Create new project
  - Select Simple Ada Project and click Next
  - Fill in a location to to deploy the project
  - Set **main name** to *say_hello* and click Apply
- Expand the **src** level in the Project View and double-click `say_hello.adb`
  - Replace the code in the file with the program shown on the previous slide
- Execute the program by selecting Build → Project → Build & Run → say_hello.adb
  - Shortcut is the ▶ in the icons bar
- Result should appear in the bottom pane labeled *Run: say_hello.exe*

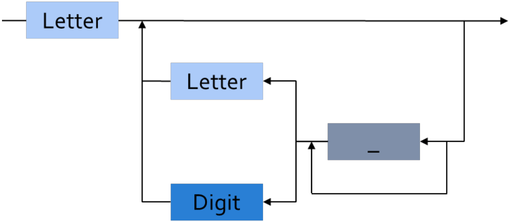
Note on GNAT File Naming Conventions

- GNAT compiler assumes one compilable entity per file
  - Package specification, subprogram body, etc
  - So the body for `say_hello` should be the only thing in the file
- Filenames should match the name of the compilable entity
  - Replacing "." with "-"
  - File extension is ".ads" for specifications and ".adb" for bodies
  - So the body for `say_hello` will be in **`say_hello.adb`**
    - If there was a specification for the subprogram, it would be in **`say_hello.ads`**
- This is the **default** behavior. There are ways around both of these rules
  - For further information, see Section 3.3 *File Naming Topics and Utilities* in the **GNAT User's Guide**

Declarations

Introduction

Identifiers



- Legal identifiers

```
Phase2
A
Space_Person
```

- Not legal identifiers

```
Phase2__1
A_
_space_person
```

String Literals

```
A_Null_String : constant string := "";
   -- two double quotes with nothing inside
String_Of_Length_One : constant string := "A";
Embedded_Single_Quotes : constant string :=
                         "Embedded 'single' quotes";
Embedded_Double_Quotes : constant string :=
                         "Embedded ""double"" quotes";
```

Identifiers, Comments, and Pragmas

Identifiers

- Syntax

```
identifier ::= letter {['_'] letter_or_digit}
```

- Character set **Unicode** 4.0
  - 8, 16, 32 bit-wide characters
- Case **not significant**
  - **SpacePerson** $\Longleftrightarrow$ **SPACEPERSON**
  - but **different** from **Space_Person**
- Reserved words are **forbidden**

Reserved Words

`abort`	`else`	`null`	`reverse`
`abs`	`elsif`	`of`	`select`
`abstract` (95)	`end`	`or`	`separate`
`accept`	`entry`	`others`	`some` (2012)
`access`	`exception`	`out`	`subtype`
`aliased` (95)	`exit`	`overriding` (2005)	`synchronized` (2005)
`all`	`for`	`package`	`tagged` (95)
`and`	`function`	`parallel` (2022)	`task`
`array`	`generic`	`pragma`	`terminate`
`at`	`goto`	`private`	`then`
`begin`	`if`	`procedure`	`type`
`body`	`in`	`protected` (95)	`until` (95)
`case`	`interface` (2005)	`raise`	`use`
`constant`	`is`	`range`	`when`
`declare`	`limited`	`record`	`while`
`delay`	`loop`	`rem`	`with`
`delta`	`mod`	`renames`	`xor`
`digits`	`new`	`requeue` (95)	
`do`	`not`	`return`	

Comments

- Terminate at end of line (i.e., no comment terminator sequence)

```
-- This is a multi-
-- line comment
A : B; -- this is an end-of-line comment
```

Pragmas

- Compiler directives
  - Compiler action *not part of* Ada grammar
  - Only **suggestions**, may be **ignored**
  - Either standard or implementation-defined
- Unrecognized pragmas
  - **No effect**
  - Cause **warning** (standard mode)
- Malformed pragmas are **illegal**

```
pragma Page;
pragma Optimize (Off);
```

Quiz

Which statement is legal?

A. `Function : constant := 1;`

B. `Fun_ction : constant := 1;`

C. `Fun_ction : constant := --initial value-- 1;`

D. `Integer Fun_ction;`

Quiz

Which statement is legal?

A. `Function : constant := 1;`
B. ***`Fun_ction : constant := 1;`***
C. `Fun_ction : constant := --initial value-- 1;`
D. `Integer Fun_ction;`

Explanations

A. **`function`** is a reserved word
B. Correct
C. Cannot have inline comments
D. C-style declaration not allowed

Numeric Literals

Decimal Numeric Literals

- Syntax

```
decimal_literal ::=
  numeral [.numeral] E [+numeral|-numeral]
numeral ::= digit {['_'] digit}
```

- Underscore is not significant
- **E** (exponent) must always be integer
- Examples

```
12      0      1E6         123_456
12.0    0.0    3.14159_26  2.3E-4
```

Based Numeric Literals

```
based_literal ::= base # numeral [.numeral] # exponent
numeral ::= base_digit { '_' base_digit }
```

- Base can be 2 .. 16
- Exponent is always a base 10 integer

```
16#FFF#           => 4095
2#1111_1111_1111# => 4095 -- With underline
16#F.FF#E+2       => 4095.0
8#10#E+3          => 4096 (8 * 8**3)
```

Comparison To C's Based Literals

- Design in reaction to C issues
- C has **limited** bases support
  - Bases 8, 10, 16
  - No base 2 in standard
- Zero-prefixed octal `0nnn`
  - **Hard** to read
  - **Error-prone**

Quiz

Which statement is legal?

A. `I : constant := 0_1_2_3_4;`
B. `F : constant := 12.;`
C. `I : constant := 8#77#E+1.0;`
D. `F : constant := 2#1111;`

Quiz

Which statement is legal?

A. *`I : constant := 0_1_2_3_4;`*
B. `F : constant := 12.;`
C. `I : constant := 8#77#E+1.0;`
D. `F : constant := 2#1111;`

Explanations

A. Underscores are not significant - they can be anywhere (except first and last character, or next to another underscore)
B. Must have digits on both sides of decimal
C. Exponents must be integers
D. Missing closing #

Object Declarations

Declarations

- Associate a *name* to an *entity*
  - Objects
  - Types
  - Subprograms
  - et cetera
- *Declaration* **must precede** use
- **Some** implicit declarations
  - **Standard** types and operations
  - **Implementation**-defined

Object Declarations

- An `object` is either *variable* or *constant*
- Basic Syntax

```
<name> : <subtype> [:= <initial value>];
<name> : constant <subtype> [:= <initial value>];
```

- Examples

```
Z, Phase : Analog;
Max : constant Integer := 200;
-- variable with a constraint
Count : Integer range 0 .. Max := 0;
-- dynamic initial value via function call
Root : Tree := F(X);
```

Multiple Object Declarations

- Allowed for convenience

```
A, B : Integer := Next_Available(X);
```

- Identical to series of single declarations

```
A : Integer := Next_Available(X);
B : Integer := Next_Available(X);
```

- Warning: may get different value

```
T1, T2 : Time := Current_Time;
```

Predefined Declarations

- **Implicit** declarations
- Language standard
- Annex A for *Core*
  - Package `Standard`
  - Standard types and operators
    - Numerical
    - Characters
  - About **half the RM** in size
- "Specialized Needs Annexes" for *optional*
- Also, implementation specific extensions

Implicit vs. Explicit Declarations

- *Explicit* → in the source

```
type Counter is range 0 .. 1000;
```

- *Implicit* → **automatically** by the compiler

```
function "+" (Left, Right : Counter) return Counter;
function "-" (Left, Right : Counter) return Counter;
function "*" (Left, Right : Counter) return Counter;
function "/" (Left, Right : Counter) return Counter;
...
```

  - Compiler creates appropriate operators based on the underlying type
    - Numeric types get standard math operators
    - Array types get concatenation operator
    - Most types get assignment operator

Elaboration

- *Elaboration* has several aspects:
- **Initial value** calculation
  - Evaluation of the expression
  - Done at **run-time** (unless static)
- Object creation
  - Memory **allocation**
  - Initial value assignment (and type checks)
- Runs in linear order
  - Follows the program text
  - Top to bottom

```
declare
  First_One : Integer := 10;
  Next_One : Integer := First_One;
  Another_One : Integer := Next_One;
begin
  ...
```

Quiz

Which block is **not** legal?

A. `A, B, C : Integer;`

B. `Integer : Standard.Integer;`

C. `Null : Integer := 0;`

D.
```
A : Integer := 123;
B : Integer := A * 3;
```

Quiz

Which block is **not** legal?

A. `A, B, C : Integer;`

B. `Integer : Standard.Integer;`

C. `Null : Integer := 0;`

D.
```
A : Integer := 123;
B : Integer := A * 3;
```

Explanations

A. Multiple objects can be created in one statement
B. `Integer` is *predefined* so it can be overridden
C. `null` is *reserved* so it can **not** be overridden
D. Elaboration happens in order, so `B` will be 369

Universal Types

Universal Types

- Implicitly defined
- Entire *classes* of numeric types
    - **universal_integer**
    - **universal_real**
    - **universal_fixed**
- Match any integer / real type respectively
    - **Implicit** conversion, as needed

```
X : Integer64 := 2;
Y : Integer8 := 2;
```

Numeric Literals Are Universally Typed

- No need to type them
  - e.g `0UL` as in C
- Compiler handles typing
  - No bugs with precision

```
X : Unsigned_Long := 0;
Y : Unsigned_Short := 0;
```

Literals Must Match "Class" of Context

- **universal_integer** literals → **Integer**
- **universal_real** literals → **fixed** or **floating** point
- Legal

```
X : Integer := 2;
Y : Float := 2.0;
```

- Not legal

```
X : Integer := 2.0;
Y : Float := 2;
```

Named Numbers

Named Numbers

- Associate a **name** with an **expression**
  - Used as **constant**
  - **universal_integer**, or **universal_real**
  - compatible with integer / real respectively
  - Expression must be **static**
- Syntax

```
<name> : constant := <static_expression>;
```

- Example

```
Pi : constant := 3.141592654;
One_Third : constant := 1.0 / 3.0;
```

A Sample Collection of Named Numbers

```
package Physical_Constants is
  Polar_Radius : constant := 20_856_010.51;
  Equatorial_Radius : constant := 20_926_469.20;
  Earth_Diameter : constant :=
    2.0 * ((Polar_Radius + Equatorial_Radius)/2.0);
  Gravity : constant := 32.1740_4855_6430_4;
  Sea_Level_Air_Density : constant :=
    0.002378;
  Altitude_Of_Tropopause : constant := 36089.0;
  Tropopause_Temperature : constant := -56.5;
end Physical_Constants;
```

Named Number Benefit

- Evaluation at **compile time**
  - As if **used directly** in the code
  - **Perfect** accuracy

```
Named_Number   : constant :=       1.0 / 3.0;
Typed_Constant : constant Float := 1.0 / 3.0;
```

Object	Named_Number	Typed_Constant
`F32 : Float_32;`	3.33333E-01	3.33333E-01
`F64 : Float_64;`	3.333333333333333E-01	3.333333_43267441E-01
`F128 : Float_128;`	3.33333333333333333E-01	3.333333_43267440796E-01

Scope and Visibility

Scope and Visibility

- *Scope* of a name
  - Where the name is **potentially** available
  - Determines **lifetime**
  - Scopes can be **nested**
- *Visibility* of a name
  - Where the name is **actually** available
  - Defined by **visibility rules**
  - **Hidden** → *in scope* but **not visible**

Introducing Block Statements

- **Sequence** of statements
  - Optional *declarative part*
  - Can be **nested**
  - Declarations **can hide** outer variables

- Syntax

```
[<block-name> :] declare
   <declarative part>
begin
   <statements>
end [block-name];
```

- Example

```
Swap: declare
  Temp : Integer;
begin
  Temp := U;
  U := V;
  V := Temp;
end Swap;
```

Scope and "Lifetime"

- Object in scope → exists
- No *scoping* keywords
  - C's **static**, **auto** etc...

```
Outer : declare
   I : Integer;
begin
   I := 1;
   Inner : declare
      F : Float;
   begin
      F := 1.0;
   end Inner;
   I := I + 1;
end Outer;
```

Scope of I

Scope of F

Name Hiding

- Caused by **homographs**
  - **Identical** name
  - **Different** entity

```
declare
  M : Integer;
begin
  M := 123;
  declare
    M : Float;
  begin
    M := 12.34; -- OK
    M := 0;     -- compile error: M is a Float
  end;
  M := 0.0; -- compile error: M is an Integer
  M := 0;   -- OK
end;
```

Overcoming Hiding

- Add a **prefix**
  - Needs named scope
- Homographs are a *code smell*
  - May need **refactoring**...

```
Outer : declare
  M : Integer;
begin
  M := 123;
  declare
    M : Float;
  begin
    M := 12.34;
    Outer.M := Integer(M);  -- reference "hidden" Integer M
  end;
end Outer;
```

Quiz

What output does the following code produce? (Assume `Print` prints the current value of its argument)

```
declare
   M : Integer := 1;
begin
   M := M + 1;
   declare
      M : Integer := 2;
   begin
      M := M + 2;
      Print (M);
   end;
   Print (M);
end;
```

A. 2, 2
B. 2, 4
C. 4, 4
D. 4, 2

Quiz

What output does the following code produce? (Assume `Print` prints the current value of its argument)

```
declare
   M : Integer := 1;
begin
   M := M + 1;
   declare
      M : Integer := 2;
   begin
      M := M + 2;
      Print (M);
   end;
   Print (M);
end;
```

A. 2, 2
B. 2, 4
C. 4, 4
D. ***4, 2***

Explanation

- Inner `M` gets printed first. It is initialized to 2 and incremented by 2
- Outer `M` gets printed second. It is initialized to 1 and incremented by 1

Aspect Clauses

Aspect Clauses

Ada 2012

- Define **additional** properties of an entity
    - Representation (eg. `with Pack`)
    - Operations (eg. `Inline`)
    - Can be **standard** or **implementation**-defined
- Usage close to pragmas
    - More **explicit**, **typed**
    - **Cannot** be ignored
    - **Recommended** over pragmas
- Syntax
    - *Note:* always part of a **declaration**

```
with aspect_mark [ => expression]
     {, aspect_mark [ => expression] }
```

Aspect Clause Example: Objects

Ada 2012

- Updated **object syntax**

```
<name> : <subtype_indication> [:= <initial value>]
               with aspect_mark [ => expression]
               {, aspect_mark [ => expression] };
```

- Usage

```
CR1 : Control_Register with
   Size    => 8,
   Address => To_Address (16#DEAD_BEEF#);

-- Prior to Ada 2012
-- using *representation clauses*
CR2 : Control_Register;
for CR2'Size use 8;
for CR2'Address use To_Address (16#DEAD_BEEF#);
```

Boolean Aspect Clauses

Ada 2012

- **Boolean** aspects only
- Longhand

```
procedure Foo with Inline => True;
```

- Aspect name only → **True**

```
procedure Foo with Inline; -- Inline is True
```

- No aspect → **False**

```
procedure Foo; -- Inline is False
```

  - Original form!

Summary

Summary

- Declarations of a **single** type, permanently
  - OOP adds flexibility
- Named-numbers
  - **Infinite** precision, **implicit** conversion
- **Elaboration** concept
  - Value and memory initialization at **run-time**
- Simple **scope** and **visibility** rules
  - **Prefixing** solves **hiding** problems
- Pragmas, Aspects
- Detailed syntax definition in Annex P (using BNF)

Basic Types

Introduction

Ada Type Model

- *Static* Typing
  - Object type **cannot change**
- *Strong* Typing
  - By **name**
  - **Compiler-enforced** operations and values
  - **Explicit** conversion for "related" types
  - **Unchecked** conversions possible

Strong Typing

- Definition of *type*
  - Applicable **values**
  - Applicable *primitive* **operations**
- Compiler-enforced
  - **Check** of values and operations
  - Easy for a computer
  - Developer can focus on **earlier** phase: requirement

A Little Terminology

- **Declaration** creates a **type name**

```
type <name> is <type definition>;
```

- **Type-definition** defines its structure
    - Characteristics, and operations
    - Base "class" of the type

```
type Type_1 is digits 12; -- floating-point
type Type_2 is range -200 .. 200; -- signed integer
type Type_3 is mod 256; -- unsigned integer
```

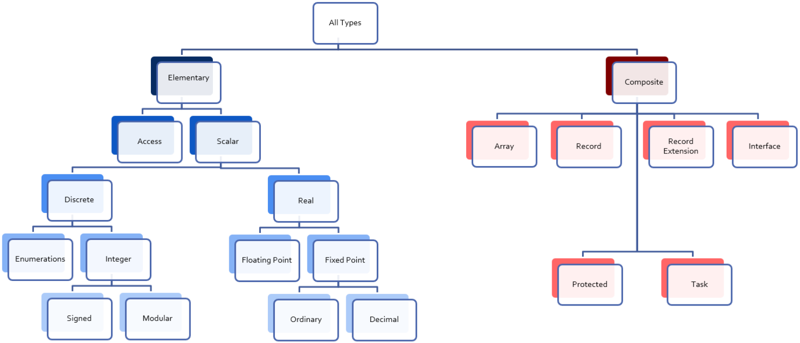
- *Representation* is the memory-layout of an **object** of the type

Ada "Named Typing"

- **Name** differentiate types
- Structure does **not**
- Identical structures may **not** be interoperable

```
type Yen is range 0 .. 100_000_000;
type Ruble is range 0 .. 100_000_000;
Mine : Yen;
Yours : Ruble;
...
Mine := Yours; -- not legal
```

Categories of Types



Scalar Types

- Indivisible: No components
- **Relational** operators defined (<, =, ...)
  - **Ordered**
- Have common **attributes**
- **Discrete** Types
  - Integer
  - Enumeration
- **Real** Types
  - Floating-point
  - Fixed-point

Discrete Types

- **Individual** ("discrete") values
  - 1, 2, 3, 4 ...
  - Red, Yellow, Green
- Integer types
  - Signed integer types
  - Modular integer types
    - Unsigned
    - **Wrap-around** semantics
    - Bitwise operations
- Enumeration types
  - Ordered list of **logical** values

Attributes

- Functions *associated* with a type
  - May take input parameters
- Some are language-defined
  - *May* be implementation-defined
  - **Built-in**
  - Cannot be user-defined
  - Cannot be modified
- See RM K.2 *Language-Defined Attributes*
- Syntax

```
Type_Name'Attribute_Name;
Type_Name'Attribute_With_Param (Param);
```

- ' often named *tick*

Discrete Numeric Types

Signed Integer Types

- Range of signed **whole** numbers
  - Symmetric about zero (-0 = +0)
- Syntax

```
type <identifier> is range  <lower> .. <upper>;
```

- Implicit numeric operators

```
-- 12-bit device
type Analog_Conversions is range 0 .. 4095;
Count : Analog_Conversions;
...
begin
   ...
   Count := Count + 1;
   ...
end;
```

Specifying Integer Type Bounds

- Must be **static**
    - Compiler selects **base type**
    - Hardware-supported integer type
    - Compilation **error** if not possible

Predefined Integer Types

- `Integer` $>=$ **16 bits** wide
- Other **probably** available
  - `Long_Integer`, `Short_Integer`, etc.
  - Guaranteed ranges: `Short_Integer` `<=` `Integer` `<=` `Long_Integer`
  - Ranges are all **implementation-defined**
- Portability not guaranteed
  - But may be difficult to avoid

Operators for Any Integer Type

- By increasing precedence

  relational operator `= | /= | < | <= | > | >=`

  binary adding operator `+ | -`

  unary adding operator `+ | -`

  multiplying operator `* | / | mod | rem`

  highest precedence operator `** | abs`

- *Note*: for exponentiation `**`
  - Result will be `Integer`
  - So power **must** be `Integer >= 0`
- Division by zero → `Constraint_Error`

Integer Overflows

- Finite binary representation
- Common source of bugs

```
K : Short_Integer := Short_Integer'Last;
...
K := K + 1;

 2#0111_1111_1111_1111#  = (2**16)-1

+                    1

======================
 2#1000_0000_0000_0000#  = -32,768
```

Integer Overflow: Ada vs others

- Ada
  - `Constraint_Error` standard exception
  - Incorrect numerical analysis
- Java
  - Silently **wraps** around (as the hardware does)
- C/C++
  - **Undefined** behavior (typically silent wrap-around)

Modular Types

- Integer type
- **Unsigned** values
- Adds operations and attributes
  - Typically **bit-wise** manipulation
- Syntax

```
type <identifier> is mod <modulus>;
```

- Modulus must be **static**
- Resulting range is 0 .. modulus-1

```
type Unsigned_Word is mod 2**16; -- 16 bits, 0..65535
type Byte is mod 256;            -- 8 bits, 0..255
```

Modular Type Semantics

- Standard `Integer` operators
- **Wraps-around** in overflow
  - Like other languages' unsigned types
  - Attributes `'Pred` and `'Succ`
- Additional bit-oriented operations are defined
  - `and`, `or`, `xor`, `not`
  - **Bit shifts**
  - Values as **bit-sequences**

Predefined Modular Types

- In `Interfaces` package
  - Need **explicit** import
- **Fixed-size** numeric types
- Common name **format**
  - `Unsigned_n`
  - `Integer_n`

```
type Integer_8 is range -2 ** 7 .. 2 ** 7 - 1;
type Integer_16 is range -2 ** 15 .. 2 ** 15 - 1;
...
type Unsigned_8 is mod 2 ** 8;
type Unsigned_16 is mod 2 ** 16;
```

String Attributes For All Scalars

- `T'Image(input)`
  - Converts `T` → `String`
- `T'Value(input)`
  - Converts `String` → `T`

```
Number : Integer := 12345;
Input  : String(1 .. N);
...
Put_Line(Integer'Image(Number));
...
Get(Input);
Number := Integer'Value(Input);
```

Range Attributes For All Scalars

- `T'First`
  - First (**smallest**) value of type `T`
- `T'Last`
  - Last (**greatest**) value of type `T`
- `T'Range`
  - Shorthand for `T'First .. T'Last`

```
type Signed_T is range -99 .. 100;
Smallest : Signed_T := Signed_T'First; -- -99
Largest  : Signed_T := Signed_T'Last;  -- 100
```

Neighbor Attributes For All Scalars

- `T'Pred (Input)`
  - Predecessor of specified value
  - Input type must be `T`
- `T'Succ (Input)`
  - Successor of specified value
  - Input type must be `T`

```
type Signed_T is range -128 .. 127;
type Unsigned_T is mod 256;
Signed   : Signed_T := -1;
Unsigned : Unsigned_T := 0;
...
Signed := Signed_T'Succ(Signed); -- Signed = 0
...
Unsigned := Unsigned_T'Pred(Unsigned); -- Signed = 255
```

Min/Max Attributes For All Scalars

- `T'Min (Value_A, Value_B)`
  - **Lesser** of two `T`
- `T'Max (Value_A, Value_B)`
  - **Greater** of two `T`

```
Safe_Lower : constant := 10;
Safe_Upper : constant := 30;
C : Integer := 15;
...
C := Integer'Max (Safe_Lower, C - 1);
...
C := Integer'Min (Safe_Upper, C + 1);
```

Quiz

What happens when you try to compile/run this code?

```
C1 : constant := 2 ** 1024;
C2 : constant := 2 ** 1024 + 10;
C3 : constant := C1 - C2;
V  : Integer := C1 - C2;
```

A. Compile error
B. Run-time error
C. V is assigned to -10
D. Unknown - depends on the compiler

Quiz

What happens when you try to compile/run this code?

```
C1 : constant := 2 ** 1024;
C2 : constant := 2 ** 1024 + 10;
C3 : constant := C1 - C2;
V  : Integer := C1 - C2;
```

A. Compile error
B. Run-time error
C. ***V is assigned to -10***
D. Unknown - depends on the compiler

Explanations

- 2^{1024} too big for most run-times BUT
- `C1`, `C2`, and `C3` are named numbers, not typed constants
  - Compiler uses unbounded precision for named numbers
  - Large intermediate representation does not get stored in object code
- For assignment to `V`, subtraction is computed by compiler
  - `V` is assigned the value -10

Enumeration Types

Enumeration Types

- Enumeration of **logical** values
    - Integer value is an implementation detail
- Syntax

```
type <identifier> is (<identifier-list>) ;
```

- Literals
    - Distinct, ordered
    - Can be in **multiple** enumerations

```
type Colors is (Red, Orange, Yellow, Green, Blue, Violet);
type Stop_Light is (Red, Yellow, Green);
...
-- Red both a member of Colors and Stop_Light
Shade : Colors := Red;
Light : Stop_Light := Red;
```

Enumeration Type Operations

- Assignment, relationals
- **Not** numeric quantities
  - *Possible* with attributes
  - Not recommended

```
type Directions is (North, South, East, West);
type Days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
Heading : Directions;
Today, Tomorrow : Days;
...
Today := Mon;
Today := North; -- compile error
Heading := South;
Heading := East + 1; -- compile error
if Today < Tomorrow then ...
```

Character Types

- Literals
  - Enclosed in single quotes eg. `'A'`
  - Case-sensitive
- **Special-case** of enumerated type
  - At least one character enumeral
- System-defined `Character`
- Can be user-defined

```
type EBCDIC is (nul, ..., 'a' , ..., 'A', ..., del);
Control : EBCDIC := 'A';
Nullo : EBCDIC := nul;
```

Language-Defined Type Boolean

- Enumeration

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

- Supports assignment, relational operators, attributes

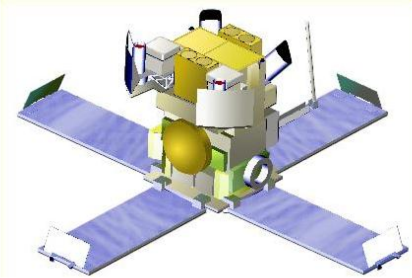
```
A : Boolean;
Counter : Integer;
...
A := (Counter = 22);
```

- Logical operators **and**, **or**, **xor**, **not**

```
A := B or (not C); -- For A, B, C boolean
```

Why Boolean Isn't Just An Integer?

- Example: Real-life error
  - HETE-2 satellite **attitude control** system software (ACS)
  - Written in **C**
- Controls four "solar paddles"
  - Deployed after launch



Why Boolean Isn't Just An Integer!

- **Initially** variable with paddles' state
  - Either **all** deployed, or **none** deployed
- Used `int` as a boolean

```
if (rom->paddles_deployed == 1)
  use_deployed_inertia_matrix();
else
  use_stowed_inertia_matrix();
```

- Later `paddles_deployed` became a **4-bits** value
  - One bit per paddle
  - `0` → none deployed, `0xF` → all deployed
- Then, `use_deployed_inertia_matrix()` if only first paddle is deployed!
- Better: boolean function `paddles_deployed()`
  - Single line to modify

Boolean Operators' Operand Evaluation

- Evaluation order **not specified**
- May be needed
  - Checking value **before** operation
  - Dereferencing null pointers
  - Division by zero

```
if Divisor /= 0 and K / Divisor = Max then ... -- Problem!
```

Short-Circuit Control Forms

- **Short-circuit** → **fixed** evaluation order
- Left-to-right
- Right only evaluated **if necessary**
  - `and then`: if left is `False`, skip right

    ```
    Divisor /= 0 and then K / Divisor = Max
    ```
  - `or else`: if left is `True`, skip right

    ```
    Divisor = 0 or else K / Divisor = Max
    ```

Quiz

```
type Enum_T is (Able, Baker, Charlie);
```

Which statement will generate an error?

A. `V1 :  Enum_T := Enum_T'Value ("Able");`
B. `V2 :  Enum_T := Enum_T'Value ("BAKER");`
C. `V3 :  Enum_T := Enum_T'Value (" charlie ");`
D. `V4 : Enum_T := Enum_T'Value ("Able Baker Charlie");`

Quiz

```
type Enum_T is (Able, Baker, Charlie);
```

Which statement will generate an error?

A. `V1 :  Enum_T := Enum_T'Value ("Able");`
B. `V2 :  Enum_T := Enum_T'Value ("BAKER");`
C. `V3 :  Enum_T := Enum_T'Value (" charlie ");`
D. *`V4 : Enum_T := Enum_T'Value ("Able Baker Charlie");`*

Explanations

A. Legal
B. Legal - conversion is case-insensitive
C. Legal - leading/trailing blanks are ignored
D. Value tries to convert entire string, which will fail at run-time

Real Types

Real Types

- Approximations to **continuous** values
  - 1.0, 1.1, 1.11, 1.111 ... 2.0, ...
  - Finite hardware $\rightarrow$ approximations
- Floating-point
  - **Variable** exponent
  - **Large** range
  - Constant **relative** precision
- Fixed-point
  - **Constant** exponent
  - **Limited** range
  - Constant **absolute** precision
  - Subdivided into Binary and Decimal
- Class focuses on floating-point

Real Type (Floating and Fixed) Literals

- **Must** contain a fractional part
- No silent promotion

```
type Phase is digits 8; -- floating-point
OK : Phase := 0.0;
Bad : Phase := 0 ; -- compile error
```

Declaring Floating Point Types

- Syntax

```
type <identifier> is
    digits <expression> [range constraint];
```

  - *digits* → **minimum** number of significant digits
  - **Decimal** digits, not bits

- Compiler choses representation
  - From **available** floating point types
  - May be **more** accurate, but not less
  - If none available → declaration is **rejected**

Predefined Floating Point Types

- Type `Float` `>= 6` digits
- Additional implementation-defined types
  - `Long_Float` `>= 11` digits
- General-purpose
- Best to **avoid** predefined types
  - Loss of **portability**
  - Easy to avoid

Floating Point Type Operators

- By increasing precedence

  relational operator `= | /= | < | >= | > | >=`

  binary adding operator `+ | -`

  unary adding operator `+ | -`

  multiplying operator `* | /`

  highest precedence operator `** | abs`

- *Note* on floating-point exponentiation `**`
  - Power must be `Integer`
    - Not possible to ask for root
    - `X**0.5` → `sqrt(x)`

Floating Point Type Attributes

- *Core* attributes

```
type My_Float is digits N;  -- N static
```

  - `My_Float'Digits`
    - Number of digits **requested** (N)
  - `My_Float'Base'Digits`
    - Number of **actual** digits
  - `My_Float'Rounding (X)`
    - Integral value nearest to `X`
    - *Note* `Float'Rounding (0.5) = 1` and `Float'Rounding (-0.5) = -1`
- Model-oriented attributes
  - Advanced machine representation of the floating-point type
  - Mantissa, strict mode

Numeric Types Conversion

- Ada's integer and real are *numeric*
  - Holding a numeric value
- Special rule: can always convert between numeric types
  - Explicitly
  - `Float` → `Integer` causes **rounding**

```
declare
   N : Integer := 0;
   F : Float := 1.5;
begin
   N := Integer (F); -- N = 2
   F := Float (N); -- F = 2.0
```

Quiz

What is the output of this code?

```
declare
   F : Float := 7.6;
   I : Integer := 10;
begin
   F := Float (Integer(F) / I);
   Put_Line (Float'Image (F));
end;
```

A. 7.6
B. Compile Error
C. 8.0
D. 0.0

Quiz

What is the output of this code?

```
declare
   F : Float := 7.6;
   I : Integer := 10;
begin
   F := Float (Integer(F) / I);
   Put_Line (Float'Image (F));
end;
```

A. 7.6
B. Compile Error
C. 8.0
D. ***0.0***

Explanations

A. Result of `F := F / Float(I);`
B. Result of `F := F / I;`
C. Result of `F := Float (Integer (F)) / Float (I);`
D. Integer value of `F` is 8. Integer result of dividing that by 10 is 0. Converting to float still gives us 0

Miscellaneous

Checked Type Conversions

- Between "closely related" types
  - Numeric types
  - Inherited types
  - Array types
- Illegal conversions **rejected**
  - Unsafe **Unchecked_Conversion** available
- Functional syntax
  - Function named using destination type name

    ```
    Target_Float := Float (Source_Integer);
    ```

  - Implicitly defined
  - **Must** be explicitly called

Default Value

Ada 2012

- Not defined by language for **scalars**
- Can be done with an **aspect clause**
  - Only during type declarations
  - <value> must be static

```
type Type_Name is <type_definition>
     with Default_Value => <value>;
```

- Example

```
type Tertiary_Switch is (Off, On, Neither)
   with Default_Value => Neither;
Implicit : Tertiary_Switch; -- Implicit = Neither
Explicit : Tertiary_Switch := Neither;
```

Simple Static Type Derivation

- New type from an existing type
  - **Limited** form of inheritance: operations
  - **Not** fully OOP
  - More details later
- Strong type benefits
  - Only **explicit** conversion possible
  - eg. `Meters` can't be set from a `Feet` value
- Syntax

```
type identifier is new Base_Type [<constraints>]
```

- Example

```
type Measurement is digits 6;
type Distance is new Measurement
      range 0.0 .. Measurement'Last;
```

Subtypes

Subtype

- May **constrain** an existing type
- Still the **same** type
- Syntax

```
subtype Defining_Identifier is Type_Name [constraints];
```

  - `Type_Name` is an existing `type` or `subtype`
- If no constraint → type alias

Subtype Example

- Enumeration type with **range** constraint

```
type Days is (Sun, Mon, Tues, Wed, Thurs, Fri, Sat);
subtype Weekdays is Days range Mon .. Fri;
Workday : Weekdays; -- type Days limited to Mon .. Fri
```

- Equivalent to **anonymous** subtype

```
Same_As_Workday : Days range Mon .. Fri;
```

Kinds of Constraints

- Range constraints on scalar types

```
subtype Positive is Integer range 1 .. Integer'Last;
subtype Natural is Integer range 0 .. Integer'Last;
subtype Weekdays is Days range Mon .. Fri;
subtype Symmetric_Distribution is
    Float range -1.0 .. +1.0;
```

- Other kinds, discussed later

Effects of Constraints

- Constraints only on values

```
type Days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
subtype Weekdays is Days range Mon .. Fri;
subtype Weekend is Days range Sat .. Sun;
```

- Functionalities are **kept**

```
subtype Positive is Integer range 1 .. Integer'Last;
P : Positive;
X : Integer := P; -- X and P are the same type
```

Assignment Respects Constraints

- RHS values must satisfy type constraints
- `Constraint_Error` otherwise

```
Q : Integer  := some_value;
P : Positive := Q; -- runtime error if Q <= 0
N : Natural  := Q; -- runtime error if Q < 0
J : Integer  := P; -- always legal
K : Integer  := N; -- always legal
```

Range Constraint Examples

```
subtype Proper_Subset is Positive range 1 .. 10;
subtype Same_Constraints is Positive
    range 1 .. Integer'Last;
subtype Letter is Character range 'A' .. 'z';
subtype Upper_Case is Letter range 'A' .. 'Z';
subtype Lower_Case is Letter range 'a' .. 'z';
subtype Null_Range is Integer
    range 1 .. 0;  -- silly when hard-coded...
-- evaluated when subtype defined, not when object declared
subtype Dynamic is Integer range Lower .. Upper;
```

Quiz

```
type Enum_T is (Sat, Sun, Mon, Tue, Wed, Thu, Fri);
subtype Enum_Sub_T is Enum_T range Mon .. Fri;
```

Which subtype definition is valid?

A. `subtype A is Enum_Sub_T range Enum_Sub_T'Pred (Enum_Sub_T'First) .. Enum_Sub_T'Last;`
B. `subtype B is range Sat .. Mon;`
C. `subtype C is Integer;`
D. `subtype D is digits 6;`

Quiz

```
type Enum_T is (Sat, Sun, Mon, Tue, Wed, Thu, Fri);
subtype Enum_Sub_T is Enum_T range Mon .. Fri;
```

Which subtype definition is valid?

A. `subtype A is Enum_Sub_T range Enum_Sub_T'Pred (Enum_Sub_T'First) .. Enum_Sub_T'Last;`
B. `subtype B is range Sat .. Mon;`
C. *`subtype C is Integer;`*
D. `subtype D is digits 6;`

Explanations

A. This generates a run-time error because the first enumeral specified is not in the range of `Enum_Sub_T`
B. Compile error - no type specified
C. Correct - standalone subtype
D. `Digits 6` is used for a type definition, not a subtype

Lab

Basic Types Lab

- Create types to handle the following concepts
  - Determining average test score
    - Number of tests taken
    - Total of all test scores
  - Number of degrees in a circle
  - Collection of colors
- Create objects for the types you've created
  - Assign initial values to the objects
  - Print the values of the objects
- Modify the objects you've created and print the new values
  - Determine the average score for all the tests
  - Add 359 degrees to the initial circle value
  - Set the color object to the value right before the last possible value

Using The "Prompts" Directory

- Course material should have a link to a `Prompts` folder
- Folder contains everything you need to get started on the lab
  - GNAT Studio project file `default.gpr`
  - Annotated / simplified source files
    - Source files are templates for lab solutions
    - Files compile as is, but don't implement the requirements
    - Comments in source files give hints for the solution
- To load prompt, either
  - From within GNAT Studio, select File → Open Project and navigate to and open the appropriate `default.gpr` **OR**
  - From a command prompt, enter

```
gnastudio -P <full path to GPR file>
```

    - If you are in the appropriate directory, and there is only one GPR file, entering `gnatstudio` will start the tool and open that project
- These prompt folders should be available for most labs

Basic Types Lab Hints

- Understand the properties of the types
  - Do you need fractions or just whole numbers?
  - What happens when you want the number to wrap?
- Predefined package **Ada.Text_IO** is handy...
  - Procedure **Put_Line** takes a **String** as the parameter
- Remember attribute **'Image** returns a **String**

```
<typemark>'Image (Object)
Object'Image
```

Basic Types Lab Solution - Declarations

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Main is

   type Number_Of_Tests_T is range 0 .. 100;
   type Test_Score_Total_T is digits 6 range 0.0 .. 10_000.0;

   type Degrees_T is mod 360;

   type Cymk_T is (Cyan, Magenta, Yellow, Black);

   Number_Of_Tests  : Number_Of_Tests_T;
   Test_Score_Total : Test_Score_Total_T;

   Angle : Degrees_T;

   Color : Cymk_T;
```

Basic Types Lab Solution - Implementation

```
begin

   -- assignment
   Number_Of_Tests   := 15;
   Test_Score_Total  := 1_234.5;
   Angle             := 180;
   Color             := Magenta;

   Put_Line (Number_Of_Tests'Image);
   Put_Line (Test_Score_Total'Image);
   Put_Line (Angle'Image);
   Put_Line (Color'Image);

   -- operations / attributes
   Test_Score_Total := Test_Score_Total / Test_Score_Total_T (Number_Of_Tests);
   Angle            := Angle + 359;
   Color            := Cymk_T'Pred (Cymk_T'Last);

   Put_Line (Test_Score_Total'Image);
   Put_Line (Angle'Image);
   Put_Line (Color'Image);

end Main;
```

Basic Types Extra Credit

- See what happens when your data is invalid / illegal
    - Number of tests = 0
    - Assign a very large number to the test score total
    - Color type only has one value
    - Add a number larger than 360 to the circle value

Summary

Benefits of Strongly Typed Numerics

- **Prevent** subtle bugs
- Cannot mix `Apples` and `Oranges`
- Force to clarify **representation** needs
    - eg. constant with or with fractional part

```
type Yen is range 0 .. 1_000_000;
type Ruble is range 0 .. 1_000_000;
Mine : Yen := 1;
Yours : Ruble := 1;
Mine := Yours; -- illegal
```

User-Defined Numeric Type Benefits

- Close to **requirements**
  - Types with **explicit** requirements (range, precision, etc.)
  - Best case: Incorrect state **not possible**
- Either implemented/respected or rejected
  - No run-time (bad) suprise
- **Portability** enhanced
  - Reduced hardware dependencies

Summary

- User-defined types and strong typing is **good**
    - Programs written in application's terms
    - Computer in charge of checking constraints
    - Security, reliability requirements have a price
    - Performance **identical**, given **same requirements**
- User definitions from existing types *can* be good
- Right **trade-off** depends on **use-case**
    - More types → more precision → less bugs
    - Storing **both** feet and meters in `Float` has caused bugs
    - More types → more complexity → more bugs
    - A `Green_Round_Object_Altitude` type is probably **never needed**
- Default initialization is **possible**
    - Use **sparingly**

Statements

Introduction

Statement Kinds

```
simple_statement ::=
  null | assignment | exit |
  goto | delay | raise |
  procedure_call | return |
  requeue | entry_call |
  abort | code

compound_statement ::=
  if | case | loop |
  block | accept | select
```

Procedure Calls (Overview)

- Procedures must be defined before they are called

```
procedure Activate (This : in out Foo;
                    Flag :        Boolean);
```

- Procedure calls are statements
  - Traditional call notation

    ```
    Activate (Idle, True);
    ```

  - "Distinguished Receiver" notation

    ```
    Idle.Activate (True);
    ```

- More details in "Subprograms" section

Block Statements

Block Statements

- Local **scope**
- Optional declarative part
- Used for
  - Temporary declarations
  - Declarations as part of statement sequence
  - Local catching of exceptions
- Syntax

```
[block-name :]
[declare <declarative part> ]
begin
   <statements>
end [block-name];
```

Block Statements Example

```
begin
   Get (V);
   Get (U);
   if U > V then -- swap them
      Swap: declare
         Temp : Integer;
      begin
         Temp := U;
         U := V;
         V := Temp;
      end Swap;
      -- Temp does not exist here
   end if;
   Print (U);
   Print (V);
end;
```

Null Statements

Null Statements

- Explicit no-op statement
- Constructs with required statement
- Explicit statements help compiler
  - Oversights
  - Editing accidents

```
case Today is
  when Monday .. Thursday =>
    Work (9.0);
  when Friday =>
    Work (4.0);
  when Saturday .. Sunday =>
    null;
end case;
```

Assignment Statements

Assignment Statements

- Syntax

  ```
  <variable> := <expression>;
  ```

- Value of expression is copied to target variable
- The type of the RHS must be same as the LHS
  - Rejected at compile-time otherwise

```
type Miles_T is range 0 .. Max_Miles;
type Km_T is range 0 .. Max_Kilometers
...
M : Miles_T := 2; -- universal integer legal for any integer
K : Km_T := 2; -- universal integer legal for any integer
M := K; -- compile error
```

Assignment Statements, Not Expressions

- Separate from expressions
  - No Ada equivalent for these:

```
int a = b = c = 1;
while (line = readline(file))
   { ...do something with line... }
```

- No assignment in conditionals
  - E.g. `if (a == 1)` compared to `if (a = 1)`

Assignable Views

- A *view* controls the way an entity can be treated
  - At different points in the program text
- The named entity must be an assignable variable
  - Thus the view of the target object must allow assignment
- Various un-assignable views
  - Constants
  - Variables of `limited` types
  - Formal parameters of mode `in`

```
Max : constant Integer := 100;
...
Max := 200; -- illegal
```

Target Variable Constraint Violations

- Prevent update to target value
    - Target is not changed at all
- May compile but will raise error at runtime
    - Predefined exception `Constraint_Error` is raised
- May be detected by compiler
    - Static value
    - Value is outside base range of type

```
Max : Integer range 1 .. 100 := 100;
...
Max := 0; -- run-time error
```

Implicit Range Constraint Checking

- The following code

```
procedure Demo is
  K : Integer;
  P : Integer range 0 .. 100;
begin
  ...
  P := K;
  ...
end Demo;
```

- Generates assignment checks similar to

```
if K < 0 or K > 100 then
  raise Constraint_Error;
else
  P := K;
end if;
```

- Run-time performance impact

Not All Assignments Are Checked

- Compilers assume variables of a subtype have appropriate values
- No check generated in this code

```
procedure Demo is
  P, K : Integer range 0 .. 100;
begin
  ...
  P := K;
  ...
end Demo;
```

Quiz

```
type One_T is range 0 .. 100;
type Two_T is range 0 .. 100;
A : constant := 100;
B : constant One_T := 99;
C : constant Two_T := 98;
X : One_T := 0;
Y : Two_T := 0;
```

Which block is **not** legal?

A.
```
X := A;
Y := A;
```
B.
```
X := B;
Y := C;
```
C.
```
X := One_T(X + C);
```
D.
```
X := One_T(Y);
Y := Two_T(X);
```

Quiz

```
type One_T is range 0 .. 100;
type Two_T is range 0 .. 100;
A : constant := 100;
B : constant One_T := 99;
C : constant Two_T := 98;
X : One_T := 0;
Y : Two_T := 0;
```

Which block is **not** legal?

A. `X := A;`
   `Y := A;`
B. `X := B;`
   `Y := C;`
C. *`X := One_T(X + C);`*
D. `X := One_T(Y);`
   `Y := Two_T(X);`

Explanations

A. Legal - `A` is an untyped constant
B. Legal - `B`, `C` are correctly typed
C. Illegal - C must be cast by itself
D. Legal - Values are typecast appropriately

Conditional Statements

If-then-else Statements

- Control flow using Boolean expressions
- Syntax

```
if <boolean expression> then -- No parentheses
   <statements>;
[else
   <statements>;]
end if;
```

- At least one statement must be supplied
  - `null` for explicit no-op

If-then-elsif Statements

- Sequential choice with alternatives
- Avoids **if** nesting
- **elsif** alternatives, tested in textual order
- **else** part still optional

```
if Valve(N) /= Closed then
  Isolate (Valve(N));
  Failure (Valve (N));
else
  if System = Off then
    Failure (Valve (N));
  end if;
end if;
```

```
if Valve(N) /= Closed then
  Isolate (Valve(N));
  Failure (Valve (N));
elsif System = Off then
  Failure (Valve (N));
end if;
```

Case Statements

- Exclusionary choice among alternatives
- Syntax

```
case <expression> is
  when <choice> => <statements>;
  { when <choice> => <statements>; }
end case;

choice ::= <expression> | <discrete range>
          | others { "|" <other choice> }
```

Simple case Statements

```
type Directions is  (Forward, Backward, Left, Right);
Direction : Directions;
...
case Direction is
  when Forward =>
    Set_Mode (Forward);
    Move (1);
  when Backward =>
    Set_Mode (Backup);
    Move (-1);
  when Left =>
    Turn (1);
  when Right =>
    Turn (-1);
end case;
```

Note: No fall-through between cases

Case Statement Rules

- More constrained than a if-elsif structure
- **All** possible values must be covered
  - Explicitly
  - ... or with `others` keyword
- Choice values cannot be given more than once (exclusive)
  - Must be known at **compile** time

Others Choice

- Choice by default
  - "everything not specified so far"
- Must be in last position

```
case Today is    -- work schedule
  when Monday =>
    Go_To (Work, Arrive=>Late, Leave=>Early);
  when Tuesday | Wednesday | Thursday => -- Several choices
    Go_To (Work, Arrive=>Early, Leave=>Late);
  when Friday =>
    Go_To (Work, Arrive=>Early, Leave=>Early);
  when others => -- weekend
    Go_To (Home, Arrive=>Day_Before, Leave=>Day_After);
end case;
```

Case Statements Range Alternatives

```
case Altitude_Ft is
  when 0 .. 9 =>
    Set_Flight_Indicator (Ground);
  when 10 .. 40_000 =>
    Set_Flight_Indicator (In_The_Air);
  when others => -- Large altitude
    Set_Flight_Indicator (Too_High);
end case;
```

Dangers of *Others* Case Alternative

- Maintenance issue: new value requiring a new alternative?
  - Compiler won't warn: `others` hides it

```
type Agencies_T is (NASA, ESA, RFSA); -- could easily grow
Bureau : Agencies_T;
...
case Bureau is
  when ESA =>
     Set_Region (Europe);
  when NASA =>
     Set_Region (America);
  when others =>
     Set_Region (Russia); -- New agencies will be Russian!
end case;
```

Quiz

```
A : Integer := 100;
B : Integer := 200;
```

Which choice needs to be modified to make a valid **if** block

A.
```
if A == B and then A != 0 then
   A := Integer'First;
   B := Integer'Last;
```

B.
```
elsif A < B then
   A := B + 1;
```

C.
```
elsif A > B then
   B := A - 1;
```

D.
```
end if;
```

Quiz

```
A : Integer := 100;
B : Integer := 200;
```

Which choice needs to be modified to make a valid **if** block

A. ***if A == B and then A != 0 then***
```
   A := Integer'First;
   B := Integer'Last;
```

B.
```
elsif A < B then
   A := B + 1;
```

C.
```
elsif A > B then
   B := A - 1;
```

D.
```
end if;
```

Explanations

- A uses the C-style equality/inequality operators
- D is legal because `else` is not required

Quiz

```
type Enum_T is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
A : Enum_T;
```

Which choice needs to be modified to make a valid `case` block

```
case A is
```

A.
```
when Sun =>
   Put_Line ("Day Off");
```

B.
```
when Mon | Fri =>
   Put_Line ("Short Day");
```

C.
```
when Tue .. Thu =>
   Put_Line ("Long Day");
```

D.
```
end case;
```

Quiz

```
type Enum_T is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
A : Enum_T;
```

Which choice needs to be modified to make a valid **case** block

```
case A is
```

A. `when Sun =>`
```
   Put_Line ("Day Off");
```

B. `when Mon | Fri =>`
```
   Put_Line ("Short Day");
```

C. `when Tue .. Thu =>`
```
   Put_Line ("Long Day");
```

D. *`end case;`*

Explanations

- Ada requires all possibilities to be covered
- Add **when others** or **when** Sat

Loop Statements

Basic Loops and Syntax

- All kind of loops can be expressed
  - Optional iteration controls
  - Optional exit statements
- Syntax

```
[<name> :] [iteration_scheme] loop
      <statements>
 end loop [<name>];
```

```
iteration_scheme ::= while <boolean expression>
                     | for <loop_parameter_specification>
                     | for <loop_iterator_specification>
```

- Example

```
Wash_Hair : loop
  Lather (Hair);
  Rinse (Hair);
end loop Wash_Hair;
```

Loop Exit Statements

- Leaves innermost loop
  - Unless loop name is specified
- Syntax

```
exit [<loop name>] [when <boolean expression>];
```

- **exit when** exits with condition

```
loop
   ...
   -- If it's time to go then exit
   exit when Time_to_Go;
   ...
end loop;
```

Exit Statement Examples

- Equivalent to C's `do while`

```
loop
  Do_Something;
  exit when Finished;
end loop;
```

- Nested named loops and exit

```
Outer : loop
  Do_Something;
  Inner : loop
    ...
    exit Outer when Finished; -- will exit all the way out
    ...
  end loop Inner;
end loop Outer;
```

While-loop Statements

- Syntax

```
while boolean_expression loop
   sequence_of_statements
end loop;
```

- Identical to

```
loop
   exit when not boolean_expression;
   sequence_of_statements
end loop;
```

- Example

```
while Count < Largest loop
  Count := Count + 2;
  Display (Count);
end loop;
```

For-loop Statements

- One low-level form
  - General-purpose (looping, array indexing, etc.)
  - Explicitly specified sequences of values
  - Precise control over sequence
- Two high-level forms
  - Ada 2012
  - Focused on objects
  - Seen later with Arrays

For in Statements

- Successive values of a **discrete** type
  - eg. enumerations values
- Syntax

```
for name in [reverse] discrete_subtype_definition loop
...
end loop;
```

- Example

```
for Day in Days_T loop
   Refresh_Planning (Day);
end loop;
```

Variable and Sequence of Values

- Variable declared implicitly by loop statement
  - Has a view as constant
  - No assignment or update possible
- Initialized as `'First`, incremented as `'Succ`
- Syntactic sugar: several forms allowed

```
-- All values of a type or subtype
for Day in Days_T loop
for Day in Days_T range Mon .. Fri -- anonymous subtype
-- Constant and variable range
for Day in Mon .. Fri loop
Today, Tomorrow : Days_T;
...
for Day in Today .. Tomorrow loop
```

Low-Level For-loop Parameter Type

- The type can be implicit
  - As long as it is clear for the compiler
  - Warning: same name can belong to several enums

```
procedure Main is
   type Color_T is (Red, White, Blue);
   type Rgb_T is (Red, Green, Blue);
begin
   for Color in Red .. Blue loop  -- which Red and Blue?
      null;
   end loop;
   for Color in Rgb_T'(Red) .. Blue loop -- OK
      null;
   end loop;
```

```
main.adb:5:21: error: ambiguous bounds in range of iteration
main.adb:5:21: error: possible interpretations:
main.adb:5:21: error: type "Rgb_T" defined at line 3
main.adb:5:21: error: type "Color_T" defined at line 2
main.adb:5:21: error: ambiguous bounds in discrete range
```

- If bounds are **universal_integer**, then type is `Integer` unless otherwise specified

```
for Idx in 1 .. 3 loop -- Idx is Integer
```

```
for Idx in Short range 1 .. 3 loop -- Idx is Short
```

Null Ranges

- *Null range* when lower bound > upper bound
  - `1 .. 0`, `Fri .. Mon`
  - Literals and variables can specify null ranges
- No iteration at all (not even one)
- Shortcut for upper bound validation

```
-- Null range: loop not entered
for Today in Fri .. Mon loop
```

Reversing Low-Level Iteration Direction

- Keyword `reverse` reverses iteration values
  - Range must still be ascending
  - Null range still cause no iteration

```
for This_Day in reverse Mon .. Fri loop
```

For-Loop Parameter Visibility

- Scope rules don't change
- Inner objects can hide outer objects

```
Block: declare
  Counter : Float := 0.0;
begin
  -- For_Loop.Counter hides Block.Counter
  For_Loop : for Counter in Integer range A .. B loop
  ...
  end loop;
end;
```

Referencing Hidden Names

- Must copy for-loop parameter to some other object if needed after the loop exits
- Use dot notation with outer scope name when hiding occurs

```
Foo:
declare
   Counter : Float := 0.0;
begin
   ...
   for Counter in Integer range 1 .. Number_Read loop
      -- set declared "Counter" to loop counter
      Foo.Counter := Float (Counter);
      ...
   end loop;
   ...
end Foo;
```

Iterations Exit Statements

- Early loop exit
- Syntax

```
exit [<loop_name>] [when <condition>]
```

- No name: Loop exited **entirely**
  - Not only current iteration

```
for K in 1 .. 1000 loop
   exit when K > F(K);
end loop;
```

- With name: Specified loop exited

```
for J in 1 .. 1000 loop
    Inner: for K in 1 .. 1000 loop
       exit Inner when K > F(K);
    end loop;
end loop;
```

For-Loop with Exit Statement Example

```
-- find position of Key within Table
Found := False;
-- iterate over Table
Search : for Index in Table'Range loop
  if Table(Index) = Key then
    Found := True;
    Position := Index;
    exit Search;
  elsif Table(Index) > Key then
    -- no point in continuing
    exit Search;
  end if;
end loop Search;
```

Quiz

```
A, B : Integer := 123;
```

Which loop block is **not** legal?

A.
```
for A in 1 .. 10 loop
  A := A + 1;
end loop;
```

B.
```
for B in 1 .. 10 loop
  Put_Line (Integer'Image (B));
end loop;
```

C.
```
for C in reverse 1 .. 10 loop
  Put_Line (Integer'Image (C));
end loop;
```

D.
```
for D in 10 .. 1 loop
  Put_Line (Integer'Image (D));
end loop;
```

.

Quiz

```
A, B : Integer := 123;
```

Which loop block is **not** legal?

A.
```
for A in 1 .. 10 loop
  A := A + 1;
end loop;
```

B.
```
for B in 1 .. 10 loop
  Put_Line (Integer'Image (B));
end loop;
```

C.
```
for C in reverse 1 .. 10 loop
  Put_Line (Integer'Image (C));
end loop;
```

D.
```
for D in 10 .. 1 loop
  Put_Line (Integer'Image (D));
end loop;
```

Explanations

A. Cannot assign to a loop parameter
B. Legal - 10 iterations
C. Legal - 10 iterations
D. Legal - 0 iterations

.

GOTO Statements

GOTO Statements

- Syntax

```
goto_statement ::= goto label;
label ::= << identifier >>
```

- Rationale
  - Historic usage
  - Arguably cleaner for some situations
- Restrictions
  - Based on common sense
  - Example: cannot jump into a **case** statement

GOTO Use

- Mostly discouraged
- May simplify control flow
- For example in-loop **continue** construct

```
loop
   -- lots of code
   ...
   goto continue;
   -- lots more code
   ...
   <<continue>>
end loop;
```

- As always maintainability beats hard set rules

Lab

Statements Lab

- Requirements
  - Create a simple algorithm to count number of hours worked in a week
    - Use **Ada.Text_IO.Get_Line** to ask user for hours worked on each day
    - Any hours over 8 gets counted as 1.5 times number of hours (e.g. 10 hours worked will get counted as 11 hours towards total)
    - Saturday hours get counted at 1.5 times number of hours
    - Sunday hours get counted at 2 times number of hours
  - Print total number of hours "worked"
- Hints
  - Use **for** loop to iterate over days of week
  - Use **if** statement to determine overtime hours
  - Use **case** statement to determine weekend bonus

Statements Lab Extra Credit

- Use an inner loop when getting hours worked to check validity
    - Less than 0 should exit outer loop
    - More than 24 should not be allowed

Statements Lab Solution

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Main is
   type Days_Of_Week_T is
     (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday);
   type Hours_Worked is digits 6;

   Total_Worked : Hours_Worked := 0.0;
   Hours_Today  : Hours_Worked;
   Overtime     : Hours_Worked;
begin
   Day_Loop :
   for Day in Days_Of_Week_T loop
      Put_Line (Day'Image);
      Input_Loop :
      loop
         Hours_Today := Hours_Worked'Value (Get_Line);
         exit Day_Loop when Hours_Today < 0.0;
         if Hours_Today > 24.0 then
            Put_Line ("I don't believe you");
         else
            exit Input_Loop;
         end if;
      end loop Input_Loop;
      if Hours_Today > 8.0 then
         Overtime := Hours_Today - 8.0;
         Hours_Today := Hours_Today + 0.5 * Overtime;
      end if;
      case Day is
         when Monday .. Friday => Total_Worked := Total_Worked + Hours_Today;
         when Saturday         => Total_Worked := Total_Worked + Hours_Today * 1.5;
         when Sunday           => Total_Worked := Total_Worked + Hours_Today * 2.0;
      end case;
   end loop Day_Loop;

   Put_Line (Total_Worked'Image);
end Main;
```

Summary

Summary

- Assignments must satisfy any constraints of LHS
    - Invalid assignments don't alter target
- Intent to do nothing must be explicitly specified
- Case statements alternatives don't fall through
- Any kind of loop can be expressed with building blocks

Array Types

Introduction

Introduction

- Traditional array concept supported to any dimension

```
declare
   type Hours is digits 6;
   type Days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
   type Schedule is array (Days) of Hours;
   Workdays : Schedule;
begin
   ...
   Workdays (Mon) := 8.5;
```

Terminology

- *Index type*
  - Specifies the values to be used to access the array components
- *Component type*
  - Specifies the type of values contained by objects of the array type
  - All components are of this same type

```
type Array_T is array (Index_T) of Component_T;
```

Array Type Index Constraints

- Must be of an integer or enumeration type
- May be dynamic
- Default to predefined **Integer**
  - Same rules as for-loop parameter default type
- Allowed to be null range
  - Defines an empty array
  - Meaningful when bounds are computed at run-time
- Used to define constrained array types

```
type Schedule is array (Days range Mon .. Fri) of Float;
type Flags_T is array (-10 .. 10) of Boolean;
```

- Or to constrain unconstrained array types

```
subtype Line is String (1 .. 80);
subtype Translation is Matrix (1..3, 1..3);
```

Run-Time Index Checking

- Array indices are checked at run-time as needed
- Invalid index values result in Constraint_Error

```
procedure Test is
  type Int_Arr is array (1..10) of Integer;
  A : Int_Arr;
  K : Integer;
begin
  A := (others => 0);
  K := FOO;
  A (K) := 42; -- runtime error if Foo returns < 1 or > 10
  Put_Line (A(K)'Image);
end Test;
```

Kinds of Array Types

- *Constrained* Array Types
  - Bounds specified by type declaration
  - **All** objects of the type have the same bounds
- *Unconstrained* Array Types
  - Bounds not constrained by type declaration
  - Objects share the type, but not the bounds
  - More flexible

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

U1 : Unconstrained (1 .. 10);
S1 : String (1 .. 50);
S2 : String (35 .. 95);
```

Constrained Array Types

Constrained Array Type Declarations

- Syntax

```
constrained_array_definition ::=
   array index_constraint of subtype_indication
index_constraint ::= (discrete_subtype_definition
   {, discrete_subtype_indication})
discrete_subtype_definition ::=
   discrete_subtype_indication | range
subtype_indication ::= subtype_mark [constraint]
range ::= range_attribute_reference |
   simple_expression .. simple_expression
```

- Examples

```
type Full_Week_T is array (Days) of Float;
type Work_Week_T is array (Days range Mon .. Fri) of Float;
type Weekdays is array (Mon .. Fri) of Float;
type Workdays is array (Weekdays'Range) of Float;
```

Multiple-Dimensioned Array Types

- Declared with more than one index definition
  - Constrained array types
  - Unconstrained array types
- Components accessed by giving value for each index

```
type Three_Dimensioned is
  array (
    Boolean,
    12 .. 50,
    Character range 'a' .. 'z')
    of Integer;
  TD : Three_Dimensioned;
  ...
begin
  TD (True, 42, 'b') := 42;
  TD (Flag, Count, Char) := 42;
```

Tic-Tac-Toe Winners Example

```
-- 9 positions on a board
type Move_Number is range 1 .. 9;
-- 8 ways to win
type Winning_Combinations is
   range 1 .. 8;
-- need 3 positions to win
type Required_Positions is
   range 1 .. 3;
Winning : constant array (
   Winning_Combinations,
   Required_Positions)
   of Move_Number := (1 => (1,2,3),
                      2 => (1,4,7),
                      ...
```

1 X	2 X	3 X
4	5	6
7	8	9

1 X	2	3
4 X	5	6
7 X	8	9

1 X	2	3
4	5 X	6
7	8	9 X

Quiz

```
type Array1_T is array (1 .. 8) of Boolean;
type Array2_T is array (0 .. 7) of Boolean;
X1, Y1 : Array1_T;
X2, Y2 : Array2_T;
```

Which statement is not legal?

A. `X1 (1) := Y1 (1);`
B. `X1 := Y1;`
C. `X1 (1) := X2 (1);`
D. `X2 := X1;`

Quiz

```
type Array1_T is array (1 .. 8) of Boolean;
type Array2_T is array (0 .. 7) of Boolean;
X1, Y1 : Array1_T;
X2, Y2 : Array2_T;
```

Which statement is not legal?

A. `X1 (1) := Y1 (1);`
B. `X1 := Y1;`
C. `X1 (1) := X2 (1);`
D. *`X2 := X1;`*

Explanations

A. Legal - elements are `Boolean`
B. Legal - object types match
C. Legal - elements are `Boolean`
D. Although the sizes are the same and the elements are the same, the type is different

Unconstrained Array Types

Unconstrained Array Type Declarations

- Do not specify bounds for objects
- Thus different objects of the same type may have different bounds
- Bounds cannot change once set
- Syntax (with simplifications)

```
unconstrained_array_definition ::=
   array (index_subtype_definition
      {, index_subtype_definition})
      of subtype_indication
index_subtype_definition ::= subtype_mark range <>
```

- Examples

```
type Index is range 1 .. Integer'Last;
type Char_Arr is array (Index range <>) of Character;
```

Supplying Index Constraints for Objects

- Bounds set by:
  - Object declaration
  - Constant's value
  - Variable's initial value
  - Further type definitions (shown later)
  - Actual parameter to subprogram (shown later)
- Once set, bounds never change

```
type Schedule is array (Days range <>) of Float;
Work : Schedule (Mon .. Fri);
All_Days : Schedule (Days);
```

Bounds Must Satisfy Type Constraints

- Must be somewhere in the range of possible values specified by the type declaration
- `Constraint_Error` otherwise

```
type Index is range 1 .. 100;
type Char_Arr is array (Index range <>) of Character;
...
Wrong : Char_Arr (0 .. 10);  -- runtime error
OK : Char_Arr (50 .. 75);
```

Null Index Range

- When 'Last of the range is smaller than 'First
  - Array is empty - no elements
- When using literals, the compiler will allow out-of-range numbers to indicate empty range
  - Provided values are within the index's base type

```
type Index_T is range 1 .. 100;

type Array_T is array (Index_T range <>) of Integer;

Typical_Empty_Array : Array_T (1 .. 0);
Weird_Empty_Array   : Array_T (123 .. -5);
Illegal_Empty_Array : Array_T (999 .. 0);
```

- When the index type is a single-valued enumerated type, no empty array is possible

"String" Types

- Language-defined unconstrained array types
  - Allow double-quoted literals as well as aggregates
  - Always have a character component type
  - Always one-dimensional
- Language defines various types
  - **String**, with **Character** as component

```
subtype Positive is Integer range 1 .. Integer'Last;
type String is array (Positive range <>) of Character;
```

  - **Wide_String**, with **Wide_Character** as component
  - **Wide_Wide_String**, with **Wide_Wide_Character** as component
    - Ada 2005 and later
- Can be defined by applications too

Application-Defined String Types

- Like language-defined string types
  - Always have a character component type
  - Always one-dimensional
- Recall character types are enumeration types with at least one character literal value

```
type Roman_Digit is ('I', 'V', 'X', 'L', 'C', 'D', 'M');
type Roman_Number is array (Positive range <>)
    of Roman_Digit;
Orwellian : constant Roman_Number := "MCMLXXXIV";
```

Specifying Constraints via Initial Value

- Lower bound is `Index_subtype'First`
- Upper bound is taken from number of items in value

```
subtype Positive is Integer range 1 .. Integer'Last;
type String is array (Positive range <>)
    of Character;
...
M : String := "Hello World!";
-- M'first is positive'first (1)

type Another_String is array (Integer range <>)
    of Character;
...
M : Another_String := "Hello World!";
-- M'first is Integer'first
```

Indefinite Types

- *Indefinite types* do not provide enough information to be instantiated
  - Size
  - Representation
- Unconstrained arrays types are indefinite
  - They do not have a definite `'Size`
- Other indefinite types exist (seen later)

No Indefinite Component Types

- Arrays: consecutive elements of the exact **same type**
- Component size must be **defined**
  - No indefinite types
  - No unconstrained types
  - Constrained subtypes allowed

```
type Good is array (1 .. 10) of String (1 .. 20); -- OK
type Bad is array (1 .. 10) of String; -- Illegal
```

Arrays of Arrays

- Allowed (of course!)
  - As long as the "component" array type is constrained
- Indexed using multiple parenthesized values
  - One per array

```
declare
   type Array_of_10 is array (1..10) of Integer;
   type Array_of_Array is array (Boolean) of Array_of_10;
   A : Array_of_Array;
begin
   ...
   A (True)(3) := 42;
```

Quiz

```
type Array_T is array (Integer range <>) of Integer;
subtype Array1_T is Array_T (1 .. 4);
subtype Array2_T is Array_T (0 .. 3);
X : Array_T  := (1, 2, 3, 4);
Y : Array1_T := (1, 2, 3, 4);
Z : Array2_T := (1, 2, 3, 4);
```

Which statement is **not** legal?

A. `X (1) := Y (1);`
B. `Y (1) := Z (1);`
C. `Y := X;`
D. `Z := X;`

Quiz

```
type Array_T is array (Integer range <>) of Integer;
subtype Array1_T is Array_T (1 .. 4);
subtype Array2_T is Array_T (0 .. 3);
X : Array_T  := (1, 2, 3, 4);
Y : Array1_T := (1, 2, 3, 4);
Z : Array2_T := (1, 2, 3, 4);
```

Which statement is **not** legal?

A. *`X (1) := Y (1);`*
B. `Y (1) := Z (1);`
C. `Y := X;`
D. `Z := X;`

Explanations

A. `Array_T` starts at `Integer'First` not `1`
B. OK, both in range
C. OK, same type and size
D. OK, same type and size

Quiz

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

O : My_Array (False .. False) := (others => True);
```

What is the value of `O (True)`?

A. `False`
B. `True`
C. None: Compilation error
D. None: Runtime error

Quiz

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

```
O : My_Array (False .. False) := (others => True);
```

What is the value of `O (True)`?

A. `False`
B. `True`
C. None: Compilation error
D. ***None: Runtime error***

`True` is not a valid index for `O`.

NB: GNAT will emit a warning by default.

Quiz

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

O : My_Array (0 .. -1) := (others => True);
```

What is the value of `O'Length`?

A. 1
B. 0
C. None: Compilation error
D. None: Runtime error

Quiz

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

O : My_Array (0 .. -1) := (others => True);
```

What is the value of `O'Length`?

A. 1
B. ***0***
C. None: Compilation error
D. None: Runtime error

When the second index is less than the first index, this is an empty array. For empty arrays, the index can be out of range for the index type.

Attributes

Array Attributes

- Return info about array index bounds

O'Length	number of array components
O'First	value of lower index bound
O'Last	value of upper index bound
O'Range	another way of saying `T'First .. T'Last`

- Meaningfully applied to constrained array types
  - Only constrained array types provide index bounds
  - Returns index info specified by the type (hence all such objects)
- Meaningfully applied to array objects
  - Returns index info for the object
  - Especially useful for objects of unconstrained array types

Attributes' Benefits

- Allow code to be more robust
  - Relationships are explicit
  - Changes are localized
- Optimizer can identify redundant checks

```
declare
   type Int_Arr is array (5 .. 15) of Integer;
   Vector : Int_Arr;
begin
   ...
   for Idx in Vector'Range loop
      Vector (Idx) := Idx * 2;
   end loop;
```

  - Compiler understands `Idx` has to be a valid index for `Vector`, so no runtime checks are necessary

Nth Dimension Array Attributes

- Attribute with **parameter**

```
T'Length (n)
T'First (n)
T'Last (n)
T'Range (n)
```

- `n` is the dimension
    - defaults to 1

```
type Two_Dimensioned is array
   (1 .. 10, 12 .. 50) of T;
TD : Two_Dimensioned;
```

- `TD'First (2) = 12`
- `TD'Last  (2) = 50`
- `TD'Length (2) = 39`
- `TD'First = TD'First (1) = 1`

Quiz

```
subtype Index1_T is Integer range 0 .. 7;
subtype Index2_T is Integer range 1 .. 8;
type Array_T is array (Index1_T, Index2_T) of Integer;
X : Array_T;
```

Which comparison is False?

A. `X'Last(2) = Index2_T'Last`
B. `X'Last(1)*X'Last(2) = X'Length(1)*X'Length(2)`
C. `X'Length(1) = X'Length(2)`
D. `X'Last(1) = 7`

Quiz

```
subtype Index1_T is Integer range 0 .. 7;
subtype Index2_T is Integer range 1 .. 8;
type Array_T is array (Index1_T, Index2_T) of Integer;
X : Array_T;
```

Which comparison is False?

A. `X'Last(2) = Index2_T'Last`
B. *`X'Last(1)*X'Last(2) = X'Length(1)*X'Length(2)`*
C. `X'Length(1) = X'Length(2)`
D. `X'Last(1) = 7`

Explanations

A. $8 = 8$
B. $7*8 /= 8*8$
C. $8 = 8$
D. $7 = 7$

Operations

Object-Level Operations

- Assignment of array objects

```
A := B;
```

- Equality and inequality

```
if A = B then
```

- Conversions

```
C := Foo (B);
```

  - Component types must be the same type
  - Index types must be the same or convertible
  - Dimensionality must be the same
  - Bounds must be compatible (not necessarily equal)

Extra Object-Level Operations

- *Only for 1-dimensional arrays!*
- Concatenation

```
type String_Type is array
  (Integer range <>) of Character;
A : constant String_Type := "foo";
B : constant String_Type := "bar";
C : constant String_Type := A & B;
-- C now contains "foobar"
```

- Comparison (for discrete component types)
  - Not for all scalars
- Logical (for `Boolean` component type)
- Slicing
  - Portion of array

Slicing

- Contiguous subsection of an array
- On any **one-dimensional** array type
  - Any component type

```
procedure Test is
  S1 : String (1 .. 9) := "Hi Adam!!";
  S2 : String := "We love    !";
begin
  S2 (9..11) := S1 (4..6);
  Put_Line (S2);
end Test;
```

Result: `We love Ada!`

Slicing With Explicit Indexes

- Imagine a requirement to have a name with two parts: first and last

```
declare
   Full_Name : String (1 .. 20);
begin
   Put_Line (Full_Name);
   Put_Line (Full_Name (1..10));  -- first half of name
   Put_Line (Full_Name (11..20)); -- second half of name
```

Slicing With Named Subtypes for Indexes

- Subtype name indicates the slice index range
  - Names for constraints, in this case index constraints
- Enhances readability and robustness

```
procedure Test is
  subtype First_Name is Positive range 1 .. 10;
  subtype Last_Name is
      Positive range First_Name'Last .. 20;
  Full_Name : String(First_Name'First..Last_Name'Last);
begin
  Put_Line(Full_Name(First_Name)); -- Full_Name(1..10)
  if Full_Name (Last_Name) = SomeString then ...
```

Dynamic Subtype Constraint Example

- Useful when constraints not known at compile-time
- Example: remove file name extension

```
File_Name
  (File_Name'First
  ..
  Index (File_Name, '.', Direction => Backward));
```

Quiz

```
type Index_T is range 1 .. 10;
type OneD_T is array (Index_T) of Boolean;
type ThreeD_T is array (Index_T, Index_T, Index_T) of OneD_T;
A : ThreeD_T;
B : OneD_T;
```

Which statement is **not** legal?

A. `B(1) := A(1,2,3)(1) or A(4,3,2)(1);`
B. `B := A(2,3,4) and A(4,3,2);`
C. `A(1,2,3..4) := A(2,3,4..5);`
D. `B(3..4) := B(4..5)`

Quiz

```
type Index_T is range 1 .. 10;
type OneD_T is array (Index_T) of Boolean;
type ThreeD_T is array (Index_T, Index_T, Index_T) of OneD_T;
A : ThreeD_T;
B : OneD_T;
```

Which statement is **not** legal?

A. `B(1) := A(1,2,3)(1) or A(4,3,2)(1);`
B. `B := A(2,3,4) and A(4,3,2);`
C. *`A(1,2,3..4) := A(2,3,4..5);`*
D. `B(3..4) := B(4..5)`

Explanations

A. All three objects are just Boolean values
B. An element of `A` is the same type as `B`
C. No slicing of multi-dimensional arrays
D. Slicing allowed on single-dimension arrays

Operations Added for Ada2012

Default Initialization for Array Types

Ada 2012

- Supports constrained and unconstrained array types
- Supports arrays of any dimensionality
  - No matter how many dimensions, there is only one component type
- Uses aspect **Default_Component_Value**

```
type Vector is array (Positive range <>) of Float
   with Default_Component_Value => 0.0;
```

  - Note that creating a large object of type `Vector` might incur a run-time cost during initialization

Two High-Level For-Loop Kinds

Ada 2012

- For arrays and containers
  - Arrays of any type and form
  - Iterable containers
    - Those that define iteration (most do)
    - Not all containers are iterable (e.g., priority queues)!
- For iterator objects
  - Known as "generalized iterators"
  - Language-defined, e.g., most container data structures
- User-defined iterators too
- We focus on the arrays/containers form for now

Array/Container For-Loops

Ada 2012

- Work in terms of elements within an object
- Syntax hides indexing/iterator controls

```
for name of [reverse] array_or_container_object loop
...
end loop;
```

- Starts with "first" element unless you reverse it
- Loop parameter name is a constant if iterating over a constant, a variable otherwise

Array Component For-Loop Example

Ada 2012

- Given an array

```
type T is array (Positive range <>) of Integer;
Primes : T := (2, 3, 5, 7, 11);
```

- Component-based looping would look like

```
for P of Primes loop
   Put_Line (Integer'Image (P));
end loop;
```

- While index-based looping would look like

```
for P in Primes'range loop
   Put_Line (Integer'Image (Primes(P)));
end loop;
```

For-Loops with Multidimensional Arrays

Ada 2012

- Same syntax, regardless of number of dimensions
- As if a set of nested loops, one per dimension
  - Last dimension is in innermost loop, so changes fastest
- In low-level format looks like

```
for each row loop
   for each column loop
      print Identity (
         row, column)
   end loop
end loop
```

```
declare
  subtype Rows is Positive;
  subtype Columns is Positive;
  type Matrix is array
      (Rows range <>,
       Columns range <>) of Float;
    Identity : constant Matrix
       (1..3, 1..3) :=
         ((1.0, 0.0, 0.0),
          (0.0, 1.0, 0.0),
          (0.0, 0.0, 1.0));
begin
  for C of Identity loop
    Put_Line (Float'Image(C));
  end loop;
```

Quiz

```
declare
   type Array_T is array (1..3, 1..3) of Integer
      with Default_Component_Value => 1;
   A : Array_T;
begin
   for I in 2 .. 3 loop
      for J in 2 .. 3 loop
         A (I, J) := I * 10 + J;
      end loop;
   end loop;
   for I of reverse A loop
      Put (I'Image);
   end loop;
end;
```

Which output is correct?

A. 1 1 1 1 22 23 1 32 33
B. 33 32 1 23 22 1 1 1 1
C. 0 0 0 0 22 23 0 32 33
D. 33 32 0 23 22 0 0 0 0

NB: Without `Default_Component_Value`, init. values are random

Quiz

```
declare
   type Array_T is array (1..3, 1..3) of Integer
      with Default_Component_Value => 1;
   A : Array_T;
begin
   for I in 2 .. 3 loop
      for J in 2 .. 3 loop
         A (I, J) := I * 10 + J;
      end loop;
   end loop;
   for I of reverse A loop
      Put (I'Image);
   end loop;
end;
```

Which output is correct?

A. 1 1 1 1 22 23 1 32 33
B. ***33 32 1 23 22 1 1 1 1***
C. 0 0 0 0 22 23 0 32 33
D. 33 32 0 23 22 0 0 0 0

Explanations

A. There is a `reverse`
B. Yes
C. Default value is 1
D. No

NB: Without `Default_Component_Value`, init. values are random

Aggregates

Aggregates

- Literals for composite types
  - Array types
  - Record types
- Two distinct forms
  - Positional
  - Named
- Syntax (simplified):

```
component_expr ::=
  expression -- Defined value
  | <>       -- Default value

array_aggregate ::= (
    {component_expr ,}                          -- Positional
  | {discrete_choice_list => component_expr,}) -- Named
  -- Default "others" indices
  [others => expression]
```

Aggregate "Positional" Form

- Specifies array component values explicitly
- Uses implicit ascending index values

```
type Days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
type Working is array (Days) of Boolean;
Week : Working;
...
-- Saturday and Sunday are False, everything else true
Week := (True, True, True, True, True, False, False);
```

Aggregate "Named" Form

- Explicitly specifies both index and corresponding component values
- Allows any order to be specified
- Ranges and choice lists are allowed (like case choices)

```
type Days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
type Working is array (Days) of Boolean;
Week : Working;
...
Week := (Sat => False, Sun => False, Mon..Fri => True);
Week := (Sat | Sun => False, Mon..Fri => True);
```

Combined Aggregate Forms Not Allowed

- Some cases lead to ambiguity, therefore never allowed for array types
- Are only allowed for record types (shown in subsequent section)

```
type Days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
type Working is array (Days) of Boolean;
Week : Working;
...
Week := (True, True, True, True, True, False, False);
Week := (Sat => False, Sun => False, Mon..Fri => True);
Week := (True, True, True, True, True,
         Sat => False, Sun => False); -- invalid
Week := (Sat | Sun => False, Mon..Fri => True);
```

Aggregates Are True Literal Values

- Used any place a value of the type may be used

```
type Schedule is array (Mon .. Fri) of Float;
Work : Schedule;
Normal : constant Schedule := (8.0, 8.0, 8.0, 8.0, 8.0);
...
Work := (8.5, 8.5, 8.5, 8.5, 6.0);
...
if Work = Normal then
...
if Work = (10.0, 10.0, 10.0, 10.0, 0.0) then -- 4-day week
```

Aggregate Consistency Rules

- Must always be complete
  - They are literals, after all
  - Each component must be given a value
  - But defaults are possible (more in a moment)
- Must provide only one value per index position
  - Duplicates are detected at compile-time
- Compiler rejects incomplete or inconsistent aggregates

```
Week := (Sat => False,
         Sun => False,
         Mon .. Fri => True,
         Wed => False);
```

"Others"

- Indicates all components not yet assigned a value
- All remaining components get this single value
- Similar to case statement's **others**
- Can be used to apply defaults too

```
type Schedule is array (Days) of Float;
Work : Schedule;
Normal : constant Schedule := (8.0, 8.0, 8.0, 8.0, 8.0,
                                others => 0.0);
```

Nested Aggregates

- For multiple dimensions
- For arrays of composite component types

```
type Matrix is array (Positive range <>,
                      Positive range <>) of Float;
Mat_4x2 : Matrix (1..4, 1..2) := (1 =>  (2.5, 3.0),
                                  2 =>  (1.5, 0.0),
                                  3 =>  (2.1, 0.0),
                                  4 =>  (9.0, 0.0));
```

Tic-Tac-Toe Winners Example

```
type Move_Number is range 1 .. 9;
-- 8 ways to win
type Winning_Combinations is range 1 .. 8;
-- need 3 places to win
type Required_Positions   is range 1 .. 3;
Winning : constant array (Winning_Combinations,
                          Required_Positions) of
   Move_Number := (-- rows
                   1 => (1, 2, 3),
                   2 => (4, 5, 6),
                   3 => (7, 8, 9),
                   -- columns
                   4 => (1, 4, 7),
                   5 => (2, 5, 8),
                   6 => (3, 6, 9),
                   -- diagonals
                   7 => (1, 5, 9),
                   8 => (3, 5, 7) );
```

Defaults Within Array Aggregates

Ada 2005

- Specified via the `box` notation
- Value for component is thus taken as for stand-alone object declaration
  - So there may or may not be a defined default!
- Can only be used with "named association" form
  - But `others` counts as named form
- Syntax

```
discrete_choice_list => <>
```

- Example

```
type Int_Arr is array (1 .. N) of Integer;
Primes : Int_Arr := (1 => 2, 2 .. N => <>);
```

Named Format Aggregate Rules

- Bounds cannot overlap
  - Index values must be specified once and only once
- All bounds must be static
  - Avoids run-time cost to verify coverage of all index values
  - Except for single choice format

```
type Float_Arr is array (Integer range <>) of Float;
Ages : Float_Arr (1 .. 10) := (1 .. 3 => X, 4 .. 10 => Y);
-- illegal: 3 and 4 appear twice
Overlap : Float_Arr (1 .. 10) := (1 .. 4 => X, 3 .. 10 => Y);
N, M, K, L : Integer;
-- illegal: cannot determine if
-- every index covered at compile time
Not_Static : Float_Arr (1 .. 10) := (M .. N => X, K .. L => Y);
-- This is legal
Values : Float_Arr (1 .. N) := (1 .. N => X);
```

Quiz

```
type Array_T is array (1 .. 5) of Integer;
X : Array_T;
J : Integer := X'First;
```

Which statement is correct?

A. `X := (1, 2, 3, 4 => 4, 5 => 5);`
B. `X := (1..3 => 100, 4..5 => -100, others => -1);`
C. `X := (J => -1, J + 1..X'Last => 1);`
D. `X := (1..3 => 100, 3..5 => 200);`

Quiz

```
type Array_T is array (1 .. 5) of Integer;
X : Array_T;
J : Integer := X'First;
```

Which statement is correct?

A. `X := (1, 2, 3, 4 => 4, 5 => 5);`
B. *`X := (1..3 => 100, 4..5 => -100, others => -1);`*
C. `X := (J => -1, J + 1..X'Last => 1);`
D. `X := (1..3 => 100, 3..5 => 200);`

Explanations

A. Cannot mix positional and named notation
B. Correct - others not needed but is allowed
C. Dynamic values must be the only choice. (This could be fixed by making J a constant.)
D. Overlapping index values (3 appears more than once)

Anonymous Array Types

Anonymous Array Types

- Array objects need not be of a named type

  `A : array (1 .. 3) of B;`
- Without a type name, no object-level operations
  - Cannot be checked for type compatibility
  - Operations on components are still ok if compatible

```
declare
-- These are not same type!
  A, B : array (Foo) of Bar;
begin
  A := B;  -- illegal
  B := A;  -- illegal
  -- legal assignment of value
  A(J) := B(K);
end;
```

Lab

Array Lab

- Requirements
  - Create an array type whose index is days of the week and each element is a number
  - Create two objects of the array type, one of which is constant
  - Perform the following operations
    - Copy the constant object to the non-constant object
    - Print the contents of the non-constant object
    - Use an array aggregate to initialize the non-constant object
    - For each element of the array, print the array index and the value
    - Move part ("source") of the non-constant object to another part ("destination"), and then clear the source location
    - Print the contents of the non-constant object
- Hints
  - When you want to combine multiple strings (which are arrays!) use the concatenation operator (**&**)
  - Slices are how you access part of an array
  - Use aggregates (either named or positional) to initialize data

Multiple Dimensions

- Requirements
  - For each day of the week, you need an array of three strings containing names of workers for that day
  - Two sets of workers: weekend and weekday, but the store is closed on Wednesday (no workers)
  - Initialize the array and then print it hierarchically

Array Lab Solution - Declarations

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Main is

   type Days_Of_Week_T is
      (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
   type Unconstrained_Array_T is
      array (Days_Of_Week_T range <>) of Natural;

   Const_Arr : constant Unconstrained_Array_T := (1, 2, 3, 4
   Array_Var : Unconstrained_Array_T (Days_Of_Week_T);

   type Name_T is array (1 .. 6) of Character;
   Weekly_Staff : array (Days_Of_Week_T, 1 .. 3) of Name_T;
```

Array Lab Solution - Implementation

```
begin
   Array_Var := Const_Arr;
   for Item of Array_Var loop
      Put_Line (Item'Image);
   end loop;
   New_Line;

   Array_Var :=
     (Mon => 111, Tue => 222, Wed => 333, Thu => 444, Fri => 555, Sat => 666,
      Sun => 777);
   for Index in Array_Var'Range loop
      Put_Line (Index'Image & " => " & Array_Var (Index)'Image);
   end loop;
   New_Line;

   Array_Var (Mon .. Wed) := Const_Arr (Wed .. Fri);
   Array_Var (Wed .. Fri) := (others => Natural'First);
   for Item of Array_Var loop
      Put_Line (Item'Image);
   end loop;
   New_Line;

   Weekly_Staff := (Mon | Tue | Thu | Fri => ("Fred  ", "Barney", "Wilma "),
                    Wed    => ("closed", "closed", "closed"),
                    others => ("Pinky ", "Inky  ", "Blinky"));

   for Day in Weekly_Staff'Range (1) loop
      Put_Line (Day'Image);
      for Staff in Weekly_Staff'Range (2) loop
         Put_Line ("  " & String (Weekly_Staff (Day, Staff)));
      end loop;
   end loop;
end Main;
```

Summary

Final Notes on Type **String**

- Any single-dimensioned array of some character type is a *string type*
  - Language defines types **String**, **Wide_String**, etc.
- Just another array type: no null termination
- Language-defined support defined in Appendix A
  - **Ada.Strings.***
  - Fixed-length, bounded-length, and unbounded-length
  - Searches for pattern strings and for characters in program-specified sets
  - Transformation (replacing, inserting, overwriting, and deleting of substrings)
  - Translation (via a character-to-character mapping)

Summary

- Any dimensionality directly supported
- Component types can be any (constrained) type
- Index types can be any discrete type
  - Integer types
  - Enumeration types
- Constrained array types specify bounds for all objects
- Unconstrained array types leave bounds to the objects
  - Thus differently-sized objects of the same type
- Default initialization for large arrays may be expensive!
- Anonymously-typed array objects used in examples for brevity but that doesn't mean you should in real programs

Record Types

Introduction

Syntax and Examples

- Syntax (simplified)

```
type T is record
   Component_Name : Type [:= Default_Value];
   ...
end record;

type T_Empty is null record;
```

- Example

```
type Record1_T is record
   Field1 : Integer;
   Field2 : Boolean;
end record;
```

- Records can be **discriminated** as well

```
type T (Size : Natural := 0) is record
   Text : String (1 .. Size);
end record;
```

Components Rules

Characteristics of Components

- **Heterogeneous** types allowed
- Referenced **by name**
- May be no components, for **empty records**
- **No** anonymous types (e.g., arrays) allowed

```
type Record_1 is record
   This_Is_Not_Legal : array (1 .. 3) of Integer;
end record;
```

- **No** constant components

```
type Record_2 is record
   This_Is_Not_Legal : constant Integer := 123;
end record;
```

- **No** recursive definitions

```
type Record_3 is record
   This_Is_Not_Legal : Record_3;
end record;
```

- **No** indefinite types

```
type Record_5 is record
   This_Is_Not_Legal : String;
   But_This_Is_Legal : String (1 .. 10);
end record;
```

Multiple Declarations

- Multiple declarations are allowed (like objects)

```
type Several is record
   A, B, C : Integer := F;
end record;
```

- Equivalent to

```
type Several is record
   A : Integer := F;
   B : Integer := F;
   C : Integer := F;
end record;
```

"Dot" Notation for Components Reference

```
type Months_T is (January, February, ..., December);
type Date is record
   Day : Integer range 1 .. 31;
   Month : Months_T;
   Year : Integer range 0 .. 2099;
end record;
Arrival : Date;
...
Arrival.Day := 27;  -- components referenced by name
Arrival.Month := November;
Arrival.Year := 1990;
```

- Can reference nested components

```
Employee
   .Birth_Date
     .Month := March;
```

Quiz

```
type Record_T is record
   -- Definition here
end record;
```

Which record definition is legal?

A. `Component_1 : array (1 .. 3) of Boolean`
B. `Component_2, Component_3 : Integer`
C. `Component_1 : Record_T`
D. `Component_1 : constant Integer := 123`

Quiz

```
type Record_T is record
   -- Definition here
end record;
```

Which record definition is legal?

A. `Component_1 : array (1 .. 3) of Boolean`
B. *`Component_2, Component_3 : Integer`*
C. `Component_1 : Record_T`
D. `Component_1 : constant Integer := 123`

A. Anonymous types not allowed
B. Correct
C. No recursive definition
D. No constant component

Quiz

```
type Cell is record
   Val : Integer;
   Message : String;
end record;
```

Is the definition legal?

A. Yes
B. No

Quiz

```
type Cell is record
   Val : Integer;
   Message : String;
end record;
```

Is the definition legal?

A. Yes
B. ***No***

A `record` definition cannot have a component of an indefinite type. `String` is indefinite if you don't specify its size.

Operations

Available Operations

- Predefined
  - Equality (and thus inequality)

    ```
    if A = B then
    ```

  - Assignment

    ```
    A := B;
    ```

- User-defined
  - Subprograms

Assignment Examples

```
declare
  type Complex is record
      Real : Float;
      Imaginary : Float;
    end record;
  ...
  Phase1 : Complex;
  Phase2 : Complex;
begin
  ...
    -- object reference
   Phase1 := Phase2;  -- entire object reference
   -- component references
   Phase1.Real := 2.5;
   Phase1.Real := Phase2.Real;
end;
```

Limited Types - Quick Intro

- A **record** type can be limited
  - And some other types, described later
- *limited* types cannot be **copied** or **compared**
  - As a result then cannot be assigned
  - May still be modified component-wise

```
type Lim is limited record
    A, B : Integer;
end record;

L1, L2 : Lim := Create_Lim (1, 2); -- Initial value OK

L1 := L2; -- Illegal
if L1 /= L2 then -- Illegal
[...]
```

Aggregates

Aggregates

- Literal values for composite types
  - As for arrays
  - Default value / selector: <>, **others**
- Can use both **named** and **positional**
  - Unambiguous
- Example:

```
(Pos_1_Value,
 Pos_2_Value,
 Component_3 => Pos_3_Value,
 Component_4 => <>, -- Default value (Ada 2005)
 others => Remaining_Value)
```

Record Aggregate Examples

```
type Color_T is (Red);
type Car_T is record
   Color    : Color_T;
   Plate_No : String (1 .. 6);
   Year     : Natural;
end record;
type Complex_T is record
   Real      : Float;
   Imaginary : Float;
end record;

declare
   Car   : Car_T     := (Red, "ABC123", Year => 2_022);
   Phase : Complex_T := (1.2, 3.4);
begin
   Phase := (Real => 5.6, Imaginary => 7.8);
end;
```

Aggregate Completeness

- All component values must be accounted for
  - Including defaults via `box`
- Allows compiler to check for missed components
- Type definition

```
type Struct is record
    A : Integer;
    B : Integer;
    C : Integer;
    D : Integer;
  end record;
S : Struct;
```

- Compiler will not catch the missing component

```
S.A := 10;
S.B := 20;
S.C := 12;
Send (S);
```

- Aggregate must be complete - compiler error

```
S := (10, 20, 12);
Send (S);
```

Named Associations

- **Any** order of associations
- Provides more information to the reader
  - Can mix with positional
- Restriction
  - Must stick with named associations **once started**

```
type Complex is record
    Real : Float;
    Imaginary : Float;
  end record;
Phase : Complex := (0.0, 0.0);
...
Phase := (10.0, Imaginary => 2.5);
Phase := (Imaginary => 12.5, Real => 0.212);
Phase := (Imaginary => 12.5, 0.212); -- illegal
```

Nested Aggregates

```
type Months_T is (January, February, ..., December);
type Date is record
   Day   : Integer range 1 .. 31;
   Month : Months_T;
   Year  : Integer range 0 .. 2099;
end record;
type Person is record
   Born : Date;
   Hair : Color;
end record;
John : Person    := ((21, November, 1990), Brown);
Julius : Person  := ((2, August, 1995), Blond);
Heather : Person := ((2, March, 1989), Hair => Blond);
Megan : Person   := (Hair => Blond,
                     Born => (16, December, 2001));
```

Aggregates with Only One Component

- **Must** use named form
- Same reason as array aggregates

```
type Singular is record
   A : Integer;
end record;

S : Singular := (3);          -- illegal
S : Singular := (3 + 1);      -- illegal
S : Singular := (A => 3 + 1); -- required
```

Aggregates with **others**

- Indicates all components not yet specified (like arrays)
- All `others` get the same value
  - They must be the **exact same** type

```
type Poly is record
   A : Float;
   B, C, D : Integer;
end record;

P : Poly := (2.5, 3, others => 0);

type Homogeneous is record
   A, B, C : Integer;
end record;

Q : Homogeneous := (others => 10);
```

Quiz

What is the result of building and running this code?

```
procedure Main is
   type Record_T is record
      A, B, C : Integer;
   end record;

   V : Record_T := (A => 1);
begin
   Put_Line (Integer'Image (V.A));
end Main;
```

A. 0
B. 1
C. Compilation error
D. Runtime error

Quiz

What is the result of building and running this code?

```
procedure Main is
   type Record_T is record
      A, B, C : Integer;
   end record;

   V : Record_T := (A => 1);
begin
   Put_Line (Integer'Image (V.A));
end Main;
```

A. 0
B. 1
C. ***Compilation error***
D. Runtime error

The aggregate is incomplete. The aggregate must specify all components. You could use box notation `(A => 1, others => <>)`

Quiz

What is the result of building and running this code?

```
procedure Main is
   type My_Integer is new Integer;
   type Record_T is record
      A, B, C : Integer;
      D : My_Integer;
   end record;

   V : Record_T := (others => 1);
begin
   Put_Line (Integer'Image (V.A));
end Main;
```

A. 0
B. 1
C. Compilation error
D. Runtime error

Quiz

What is the result of building and running this code?

```
procedure Main is
   type My_Integer is new Integer;
   type Record_T is record
      A, B, C : Integer;
      D : My_Integer;
   end record;

   V : Record_T := (others => 1);
begin
   Put_Line (Integer'Image (V.A));
end Main;
```

A. 0
B. 1
C. *Compilation error*
D. Runtime error

All components associated to a value using `others` must be of the same `type`.

Quiz

```
type Nested_T is record
   Field : Integer;
end record;
type Record_T is record
   One   : Integer;
   Two   : Character;
   Three  : Integer;
   Four  : Nested_T;
end record;
X, Y : Record_T;
Z    : constant Nested_T := (others => -1);
```

Which assignment(s) is(are) **not** legal?

A. `X := (1, '2', Three => 3, Four => (6))`
B. `X := (Two => '2', Four => Z, others => 5)`
C. `X := Y`
D. `X := (1, '2', 4, (others => 5))`

Quiz

```
type Nested_T is record
   Field : Integer;
end record;
type Record_T is record
   One    : Integer;
   Two    : Character;
   Three  : Integer;
   Four   : Nested_T;
end record;
X, Y : Record_T;
Z     : constant Nested_T := (others => -1);
```

Which assignment(s) is(are) **not** legal?

A. *`X := (1, '2', Three => 3, Four => (6))`*
B. `X := (Two => '2', Four => Z, others => 5)`
C. `X := Y`
D. `X := (1, '2', 4, (others => 5))`

A. Four **must** use named association
B. `others` valid: `One` and `Three` are `Integer`
C. Valid but `Two` is not initialized
D. Positional for all components

Default Values

Component Default Values

```
type Complex is
  record
    Real : Float := 0.0;
    Imaginary : Float := 0.0;
  end record;
-- all components use defaults
Phasor : Complex;
-- all components must be specified
I : constant Complex := (0.0, 1.0);
```

Default Component Value Evaluation

- Occurs when object is elaborated
  - Not when the type is elaborated
- Not evaluated if explicitly overridden

```
type Structure is
  record
    A : Integer;
    R : Time := Clock;
  end record;
-- Clock is called for S1
S1 : Structure;
-- Clock is not called for S2
S2 : Structure := (A => 0, R => Yesterday);
```

Defaults Within Record Aggregates

Ada 2005

- Specified via the *box* notation
- Value for the component is thus taken as for a stand-alone object declaration
  - So there may or may not be a defined default!
- Can only be used with "named association" form
  - But can mix forms, unlike array aggregates

```
type Complex is
  record
    Real : Float := 0.0;
    Imaginary : Float := 0.0;
  end record;
Phase := (42.0, Imaginary => <>);
```

Default Initialization Via Aspect Clause

Ada 2012

- Not definable for entire record type
- Components of scalar types take type's default if no explicit default value specified by record type

```
type Toggle_Switch is (Off, On)
   with Default_Value => Off;
type Controller is record
   -- Off unless specified during object initialization
   Override : Toggle_Switch;
   -- default for this component
   Enable : Toggle_Switch := On;
  end record;
C : Controller; -- Override => off, Enable => On
D : Controller := (On, Off); -- All defaults replaced
```

Quiz

Ada 2012

```
function Next return Natural; -- returns next number starting with 1

type Record_T is record
   A, B : Integer := Next;
   C    : Integer := Next;
end record;
R : Record_T := (C => 100, others => <>);
```

What is the value of R?

A. (1, 2, 3)
B. (1, 1, 100)
C. (1, 2, 100)
D. (100, 101, 102)

Quiz

Ada 2012

```
function Next return Natural; -- returns next number starting with 1

type Record_T is record
   A, B : Integer := Next;
   C    : Integer := Next;
end record;
R : Record_T := (C => 100, others => <>);
```

What is the value of R?

A. (1, 2, 3)
B. (1, 1, 100)
C. ***(1, 2, 100)***
D. (100, 101, 102)

Explanations

A. `C => 100`
B. Multiple declaration calls `Next` twice
C. Correct
D. `C => 100` has no effect on `A` and `B`

Discriminated Records

Discriminated Record Types

- *Discriminated record* type
  - Different **objects** may have **different** components
  - All object **still** share the same type
- Kind of *storage overlay*
  - Similar to `union` in C
  - But preserves **type checking**
  - And object size **is related to** discriminant
- Aggregate assignment is allowed

Discriminants

```
2  type Person_Group is (Student, Faculty);
3  type Person (Group : Person_Group) is record
4     Age : Positive;
5     case Group is
6        when Student => -- 1st variant
7           Gpa  : Float range 0.0 .. 4.0;
8        when Faculty => -- 2nd variant
9           Pubs : Positive;
10    end case;
11 end record;
```

- Group (on line 3) is the *discriminant*
- Run-time check for component **consistency**
  - eg `A_Person.Pubs := 1` checks `A_Person.Group = Faculty`
  - `Constraint_Error` if check fails
- Discriminant is **constant**
  - Unless object is **mutable**
- Discriminant can be used in *variant part* (line 5)
  - Similar to case statements (all values must be covered)
  - Fields listed will only be visible if choice matches discriminant
  - Field names need to be unique (even across discriminants)
  - Variant part must be end of record (hence only one variant part allowed)

Semantics

- Person objects are **constrained** by their discriminant
  - They are **indefinite**
  - **Unless** mutable
  - Assignment from same variant **only**
  - **Representation** requirements

```
Pat  : Person (Student); -- No Pat.Pubs
Prof : Person (Faculty); -- No Prof.GPA
Soph : Person := (Group  => Student,
                  Age => 21,
                  GPA  => 3.2);
X : Person;  -- Illegal: must specify discriminant
```

```
Pat  := Soph; -- OK
Soph := Prof; -- Constraint_Error at run time
```

Mutable Discriminated Record

- When discriminant has a **default value**
  - Objects instantiated **using the default** are **mutable**
  - Objects specifying an **explicit** value are **not** mutable
  - Type is now **definite**
- Mutable records have **variable** discriminants
- Use **same** storage for **several** variant

```
-- Potentially mutable
type Person (Group : Person_Group := Student) is record

--  Use default value: mutable
S : Person;
--  Explicit value: *not* mutable
--  even if Student is also the default
S2 : Person (Group => Student);
...
S := (Group => Student, Age => 22, Gpa => 0.0);
S := (Group => Faculty, Age => 35, Pubs => 10);
```

Quiz

```
type T (Sign : Integer) is record
    case Sign is
    when Integer'First .. -1 =>
        I : Integer;
        B : Boolean;
    when others =>
        N : Natural;
    end case;
end record;

O : T (1);
```

Which component does `O` contain?

A. `O.I, O.B`
B. `O.N`
C. None: Compilation error
D. None: Runtime error

Quiz

```
type T (Sign : Integer) is record
    case Sign is
    when Integer'First .. -1 =>
        I : Integer;
        B : Boolean;
    when others =>
        N : Natural;
    end case;
end record;
```

```
O : T (1);
```

Which component does `O` contain?

A. `O.I, O.B`
B. *`O.N`*
C. None: Compilation error
D. None: Runtime error

Quiz

```
type T (Floating : Integer) is record
    case Floating is
        when 0 =>
            I : Integer;
        when 1 =>
            F : Float;
    end case;
end record;

O : T (1);
```

Which component does `O` contain?

A. `O.F, O.I`
B. `O.F`
C. None: Compilation error
D. None: Runtime error

Quiz

```
type T (Floating : Integer) is record
    case Floating is
        when 0 =>
            I : Integer;
        when 1 =>
            F : Float;
    end case;
end record;

O : T (1);
```

Which component does `O` contain?

A. `O.F, O.I`
B. `O.F`
C. ***None: Compilation error***
D. None: Runtime error

The variant `case` must cover all the possible values of `Integer`.

Quiz

```
type T (Floating : Boolean) is record
    case Floating is
        when False =>
            I : Integer;
        when True =>
            F : Float;
    end case;
    I2 : Integer;
end record;

O : T (True);
```

Which component does `O` contain?

A. `O.F, O.I2`
B. `O.F`
C. None: Compilation error
D. None: Runtime error

Quiz

```
type T (Floating : Boolean) is record
    case Floating is
        when False =>
            I : Integer;
        when True =>
            F : Float;
    end case;
    I2 : Integer;
end record;

O : T (True);
```

Which component does `O` contain?

A. `O.F, O.I2`
B. `O.F`
C. ***None: Compilation error***
D. None: Runtime error

The variant part cannot be followed by a component declaration (`I2 : Integer` there)

Lab

Record Types Lab

- Requirements
  - Create a simple First-In/First-Out (FIFO) queue record type and object
  - Allow the user to:
    - Add ("push") items to the queue
    - Remove ("pop") the next item to be serviced from the queue (Print this item to ensure the order is correct)
  - When the user is done manipulating the queue, print out the remaining items in the queue
- Hints
  - Queue record should at least contain:
    - Array of items
    - Index into array where next item will be added

Record Types Lab Solution - Declarations

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Main is

   type Name_T is array (1 .. 6) of Character;
   type Index_T is range 0 .. 1_000;
   type Queue_T is array (Index_T range 1 .. 1_000) of Name_T;

   type Fifo_Queue_T is record
      Next_Available : Index_T := 1;
      Last_Served    : Index_T := 0;
      Queue          : Queue_T := (others => (others => ' '));
   end record;

   Queue : Fifo_Queue_T;
   Choice : Integer;
```

Record Types Lab Solution - Implementation

```
begin

   loop
      Put ("1 = add to queue | 2 = remove from queue | others => done: ");
      Choice := Integer'Value (Get_Line);
      if Choice = 1 then
         Put ("Enter name: ");
         Queue.Queue (Queue.Next_Available) := Name_T (Get_Line);
         Queue.Next_Available               := Queue.Next_Available + 1;
      elsif Choice = 2 then
         if Queue.Next_Available = 1 then
            Put_Line ("Nobody in line");
         else
            Queue.Last_Served := Queue.Last_Served + 1;
            Put_Line ("Now serving: " & String (Queue.Queue (Queue.Last_Served)));
         end if;
      else
         exit;
      end if;
      New_Line;
   end loop;

   Put_Line ("Remaining in line: ");
   for Index in Queue.Last_Served + 1 .. Queue.Next_Available - 1 loop
      Put_Line ("  " & String (Queue.Queue (Index)));
   end loop;

end Main;
```

Summary

Summary

- Heterogeneous types allowed for components
- Default initial values allowed for components
  - Evaluated when each object elaborated, not the type
  - Not evaluated if explicit initial value specified
- Aggregates express literals for composite types
  - Can mix named and positional forms

Subprograms

Introduction

Introduction

- Are syntactically distinguished as `function` and `procedure`
  - Functions represent *values*
  - Procedures represent *actions*

```
function Is_Leaf (T : Tree) return Boolean
procedure Split (T : in out Tree;
                 Left : out Tree;
                 Right : out Tree)
```

- Provide direct syntactic support for separation of specification from implementation

```
function Is_Leaf (T : Tree) return Boolean;
function Is_Leaf (T : Tree) return Boolean is
begin
...
end Is_Leaf;
```

Recognizing Procedures and Functions

- Functions' results must be treated as values
    - And cannot be ignored
- Procedures cannot be treated as values
- You can always distinguish them via the call context

```
Open (Source, "SomeFile.txt");
while not End_of_File (Source) loop
  Get (Next_Char, From => Source);
  if Found (Next_Char, Within => Buffer) then
    Display (Next_Char);
  end if;
end loop;
```

A Little "Preaching" About Names

- Procedures are abstractions for actions
- Functions are abstractions for values
- Use names that reflect those facts!
    - Imperative verbs for procedure names
    - Nouns for function names, as for mathematical functions
        - Questions work for boolean functions

```
procedure Open (V : in out Valve);
procedure Close (V : in out Valve);
function Square_Root (V: Float) return Float;
function Is_Open (V: Valve) return Boolean;
```

Syntax

Specification and Body

- Subprogram specification is the external (user) **interface**
  - **Declaration** and **specification** are used synonymously
- Specification may be required in some cases
  - eg. recursion
- Subprogram body is the **implementation**

Procedure Specification Syntax (Simplified)

```
procedure Swap (A, B : in out Integer);
```

```
procedure_specification ::=
   procedure program_unit_name
     (parameter_specification
     { ; parameter_specification});

parameter_specification ::=
   identifier_list : mode subtype_mark [ := expression ]

mode ::= [in] | out | in out
```

Function Specification Syntax (Simplified)

```
function F (X : Float) return Float;
```

- Close to `procedure` specification syntax
  - With `return`
  - Can be an operator: `+ - * / mod rem` ...

```
function_specification ::=
   function designator
     (parameter_specification
     { ; parameter_specification})
     return result_type;

designator ::= program_unit_name | operator_symbol
```

Body Syntax

```
subprogram_specification is
   [declarations]
begin
   sequence_of_statements
end [designator];
```

```
procedure Hello is
begin
   Ada.Text_IO.Put_Line ("Hello World!");
   Ada.Text_IO.New_Line (2);
end Hello;

function F (X : Float) return Float is
   Y : constant Float := X + 3.0;
begin
   return X * Y;
end F;
```

Completions

- Bodies **complete** the specification
  - There are **other** ways to complete
- Separate specification is **not required**
  - Body can act as a specification
- A declaration and its body must **fully** conform
  - Mostly **semantic** check
  - But parameters **must** have same name

```
procedure P (J, K : Integer)
procedure P (J : Integer; K : Integer)
procedure P (J, K : in Integer)
-- Invalid
procedure P (A : Integer; B : Integer)
```

Completion Examples

- Specifications

```
procedure Swap (A, B : in out Integer);
function Min (X, Y : Person) return Person;
```

- Completions

```
procedure Swap (A, B : in out Integer) is
  Temp : Integer := A;
begin
  A := B;
  B := Temp;
end Swap;

-- Completion as specification
function Less_Than (X, Y : Person) return Boolean is
begin
   return X.Age < Y.Age;
end Less_Than;

function Min (X, Y : Person) return Person is
begin
   if Less_Than (X, Y) then
      return X;
   else
      return Y;
   end if;
end Min;
```

Direct Recursion - No Declaration Needed

- When **is** is reached, the subprogram becomes **visible**
  - It can call **itself** without a declaration

```
type Vector_T is array (Natural range <>) of Integer;
Empty_Vector : constant Vector_T (1 .. 0) := (others => 0);

function Get_Vector return Vector_T is
  Next : Integer;
begin
  Get (Next);

  if Next = 0 then
    return Empty_Vector;
  else
    return Get_Vector & Next;
  end if;
end Input;
```

Indirect Recursion Example

- Elaboration in **linear order**

```
procedure P;

procedure F is
begin
  P;
end F;

procedure P is
begin
  F;
end P;
```

Quiz

Which profile is semantically different from the others?

A. `procedure P (A : Integer; B : Integer);`
B. `procedure P (A, B : Integer);`
C. `procedure P (B : Integer; A : Integer);`
D. `procedure P (A : in Integer; B : in Integer);`

Quiz

Which profile is semantically different from the others?

A. `procedure P (A : Integer; B : Integer);`
B. `procedure P (A, B : Integer);`
C. ***`procedure P (B : Integer; A : Integer);`***
D. `procedure P (A : in Integer; B : in Integer);`

Parameter names are important in Ada. The other selections have the names in the same order with the same mode and type.

Parameters

Subprogram Parameter Terminology

- *Actual parameters* are values passed to a call
  - Variables, constants, expressions
- *Formal parameters* are defined by specification
  - Receive the values passed from the actual parameters
  - Specify the types required of the actual parameters
  - Type **cannot** be anonymous

```
procedure Something (Formal1 : in Integer);

ActualX : Integer;
...
Something (ActualX);
```

Parameter Associations In Calls

- Associate formal parameters with actuals
- Both positional and named association allowed

```
Something (ActualX, Formal2 => ActualY);
Something (Formal2 => ActualY, Formal1 => ActualX);
```

- Having named **then** positional is forbidden

```
-- Compilation Error
Something (Formal1 => ActualX, ActualY);
```

Actual Parameters Respect Constraints

- Must satisfy any constraints of formal parameters
- `Constraint_Error` otherwise

```
declare
  Q : Integer := ...
  P : Positive := ...
  procedure Foo (This : Positive);
begin
  Foo (Q); -- runtime error if Q <= 0
  Foo (P);
```

Parameter Modes and Return

- Mode `in`
    - Formal parameter is `constant`
        - So actual is not modified either
    - Can have **default**, used when **no value** is provided

```
procedure P (N : in Integer := 1; M : in Positive);
[...]
P (M => 2);
```

- Mode `out`
    - Writing is **expected**
    - Reading is **allowed**
    - Actual **must** be a writable object
- Mode `in out`
    - Actual is expected to be **both** read and written
    - Actual **must** be a writable object
- Function `return`
    - **Must** always be handled

Why Read Mode **out** Parameters?

- **Convenience** of writing the body
  - No need for readable temporary variable
- Warning: initial value is **not defined**

```
procedure Compute (Value : out Integer) is
begin
  Value := 0;
  for K in 1 .. 10 loop
    Value := Value + K; -- this is a read AND a write
  end loop;
end Compute;
```

Parameter Passing Mechanisms

- *By-Copy*
  - The formal denotes a separate object from the actual
  - `in`, `in out`: actual is copied into the formal **on entry to** the subprogram
  - `out`, `in out`: formal is copied into the actual **on exit from** the subprogram
- *By-Reference*
  - The formal denotes a view of the actual
  - Reads and updates to the formal directly affect the actual
  - More efficient for large objects
- Parameter **types** control mechanism selection
  - Not the parameter **modes**
  - Compiler determines the mechanism

By-Copy vs By-Reference Types

- By-Copy
  - Scalar types
  - `access` types
- By-Reference
  - `tagged` types
  - `task` types and `protected` types
  - `limited` types
- `array`, `record`
  - By-Reference when they have by-reference **components**
  - By-Reference for **implementation-defined** optimizations
  - By-Copy otherwise
- `private` depends on its full definition

Unconstrained Formal Parameters or Return

- Unconstrained **formals** are allowed
  - Constrained by **actual**
- Unconstrained `return` is allowed too
  - Constrained by the **returned object**

```
type Vector is array (Positive range <>) of Float;
procedure Print (Formal : Vector);

Phase : Vector (X .. Y);
State : Vector (1 .. 4);
...
begin
  Print (Phase);          -- Formal'Range is X .. Y
  Print (State);          -- Formal'Range is 1 .. 4
  Print (State (3 .. 4)); -- Formal'Range is 3 .. 4
```

Unconstrained Parameters Surprise

- Assumptions about formal bounds may be **wrong**

```
type Vector is array (Positive range <>) of Float;
function Subtract (Left, Right : Vector) return Vector;

V1 : Vector (1 .. 10); -- length = 10
V2 : Vector (15 .. 24); -- length = 10
R : Vector (1 .. 10); -- length = 10
...
-- What are the indices returned by Subtract?
R := Subtract (V2, V1);
```

Naive Implementation

- **Assumes** bounds are the same everywhere
- Fails when `Left'First /= Right'First`
- Fails when `Left'First /= 1`

```
function Subtract (Left, Right : Vector)
  return Vector is
   Result : Vector (1 .. Left'Length);
begin
   ...
   for K in Result'Range loop
     Result (K) := Left (K) - Right (K);
   end loop;
```

Correct Implementation

- Covers **all** bounds
- `return` indexed by `Left'Range`

```
function Subtract (Left, Right : Vector) return Vector is
   Result : Vector (Left'Range);
   Offset : constant Integer := Right'First - Result'First;
begin
   ...
   for K in Result'Range loop
     Result (K) := Left (K) - Right (K + Offset);
   end loop;
```

Quiz

```
function F (P1 : in     Integer   := 0;
            P2 : in out Integer;
            P3 : in     Character := ' ';
            P4 :    out Character)
   return Integer;
J1, J2 : Integer;
C : Character;
```

Which call is legal?

A. `J1 := F (P1 => 1, P2 => J2, P3 => '3', P4 => '4');`
B. `J1 := F (P1 => 1, P3 => '3', P4 => C);`
C. `J1 := F (1, J2, '3', C);`
D. `F (J1, J2, '3', C);`

Quiz

```
function F (P1 : in     Integer   := 0;
            P2 : in out Integer;
            P3 : in     Character := ' ';
            P4 :    out Character)
   return Integer;
J1, J2 : Integer;
C : Character;
```

Which call is legal?

A. `J1 := F (P1 => 1, P2 => J2, P3 => '3', P4 => '4');`
B. `J1 := F (P1 => 1, P3 => '3', P4 => C);`
C. *`J1 := F (1, J2, '3', C);`*
D. `F (J1, J2, '3', C);`

Explanations

A. P4 is `out`, it **must** be a variable
B. P2 has no default value, it **must** be specified
C. Correct
D. F is a function, its `return` **must** be handled

Null Procedures

Null Procedure Declarations

Ada 2005

- Shorthand for a procedure body that does nothing
- Longhand form

```
procedure NOP is
begin
  null;
end NOP;
```

- Shorthand form

```
procedure NOP is null;
```

- The `null` statement is present in both cases
- Explicitly indicates nothing to be done, rather than an accidental removal of statements

Null Procedures As Completions

Ada 2005

- Completions for a distinct, prior declaration

```
procedure NOP;
...
procedure NOP is null;
```

- A declaration and completion together
  - A body is then not required, thus not allowed

```
procedure NOP is null;
...
procedure NOP is -- compile error
begin
  null;
end NOP;
```

Typical Use for Null Procedures: OOP

Ada 2005

- When you want a method to be concrete, rather than abstract, but don't have anything for it to do
    - The method is then always callable, including places where an abstract routine would not be callable
    - More convenient than full null-body definition

Null Procedure Summary

Ada 2005

- Allowed where you can have a full body
  - Syntax is then for shorthand for a full null-bodied procedure
- Allowed where you can have a declaration!
  - Example: package declarations
  - Syntax is shorthand for both declaration and completion
    - Thus no body required/allowed
- Formal parameters are allowed

```
procedure Do_Something (P : in     Integer) is null;
```

Nested Subprograms

Subprograms within Subprograms

- Subprograms can be placed in any declarative block
  - So they can be nested inside another subprogram
  - Or even within a `declare` block
- Useful for performing sub-operations without passing parameter data

Nested Subprogram Example

```
procedure Main is

   function Read (Prompt : String) return Types.Line_T is
   begin
      Put (Prompt & "> ");
      return Types.Line_T'Value (Get_Line);
   end Read;

   Lines : Types.Lines_T (1 .. 10);
begin
   for J in Lines'Range loop
      Lines (J) := Read ("Line " & J'Image);
   end loop;
```

Procedure Specifics

Return Statements In Procedures

- Returns immediately to caller
- Optional
  - Automatic at end of body execution
- Fewer is traditionally considered better

```
procedure P is
begin
   ...
   if Some_Condition then
     return; -- early return
   end if;
   ...
end P; -- automatic return
```

Function Specifics

Return Statements In Functions

- Must have at least one
  - Compile-time error otherwise
  - Unless doing machine-code insertions
- Returns a value of the specified (sub)type
- Syntax

```
function defining_designator [formal_part]
     return subtype_mark is
   declarative_part
   begin
      {statements}
      return expression;
   end designator;
```

No Path Analysis Required By Compiler

- Running to the end of a function without hitting a **return** statement raises `Program_Error`
- Compilers can issue warning if they suspect that a **return** statement will not be hit

```
function Greater (X, Y : Integer) return Boolean is
begin
  if X > Y then
    return True;
  end if;
end Greater; -- possible compile warning
```

Multiple Return Statements

- Allowed
- Sometimes the most clear

```
function Truncated (R : Float) return Integer is
  Converted : Integer := Integer (R);
begin
  if R - Float (Converted) < 0.0 then -- rounded up
    return Converted - 1;
  else -- rounded down
    return Converted;
  end if;
end Truncated;
```

Multiple Return Statements Versus One

- Many can detract from readability
- Can usually be avoided

```
function Truncated (R : Float) return Integer is
  Result : Integer := Integer (R);
begin
  if R - Float (Result) < 0.0 then -- rounded up
    Result := Result - 1;
  end if;
  return Result;
end Truncated;
```

Function Dynamic-Size Results

```
function Char_Mult (C : Character; L : Natural)
  return String is
   R : String (1 .. L) := (others => C);
begin
   return R;
end Char_Mult;

X : String := Char_Mult ('x', 4);

begin
   -- OK
   pragma Assert (X'Length = 4 and X = "xxxx");
```

Expression Functions

Expression Functions

Ada 2012

- Functions whose implementations are pure expressions
  - No other completion is allowed
  - No `return` keyword
- May exist only for sake of pre/postconditions

```
function function_specification is (expression);
```

NB: Parentheses around expression are **required**

- Can complete a prior declaration

```
function Squared (X : Integer) return Integer;
function Squared (X : Integer) return Integer is
   (X ** 2);
```

Expression Functions Example

Ada 2012

- Expression function

```
function Square (X : Integer) return Integer is (X ** 2);
```

- Is equivalent to

```
function Square (X : Integer) return Integer is
begin
   return X ** 2;
end Square;
```

Quiz

Which statement is True?

A. Expression functions cannot be nested functions.
B. Expression functions require a specification and a body.
C. Expression functions must have at least one "return" statement.
D. Expression functions can have "out" parameters.

Quiz

Which statement is True?

A. Expression functions cannot be nested functions.
B. Expression functions require a specification and a body.
C. Expression functions must have at least one "return" statement.
D. ***Expression functions can have "out" parameters.***

Explanations

A. False, they can be declared just like regular function
B. False, an expression function cannot have a body
C. False, expression functions cannot contain a no `return`
D. Correct, but it can assign to `out` parameters only by calling another function.

Potential Pitfalls

Mode **out** Risk for Scalars

- Always assign value to `out` parameters
- Else "By-copy" mechanism will copy something back
    - May be junk
    - `Constraint_Error` or unknown behaviour further down

```
procedure P
  (A, B : in Some_Type; Result : out Scalar_Type) is
begin
  if Some_Condition then
    return;  -- Result not set
  end if;
  ...
  Result := Some_Value;
end P;
```

"Side Effects"

- Any effect upon external objects or external environment
  - Typically alteration of non-local variables or states
  - Can cause hard-to-debug errors
  - Not legal for `function` in SPARK
- Can be there for historical reasons
  - Or some design patterns

```
Global : Integer := 0;

function F (X : Integer) return Integer is
begin
   Global := Global + X;
   return Global;
end F;
```

Order-Dependent Code And Side Effects

```
Global : Integer := 0;

function Inc return Integer is
begin
  Global := Global + 1;
  return Global;
end Inc;

procedure Assert_Equals (X, Y : in Integer);
...
Assert_Equals (Global, Inc);
```

- Language does **not** specify parameters' order of evaluation
- `Assert_Equals` could get called with
  - X → 0, Y → 1 (if `Global` evaluated first)
  - X → 1, Y → 1 (if `Inc` evaluated first)

Parameter Aliasing

- *Aliasing*: Multiple names for an actual parameter inside a subprogram body
- Possible causes:
  - Global object used is also passed as actual parameter
  - Same actual passed to more than one formal
  - Overlapping `array` slices
  - One actual is a component of another actual
- Can lead to code dependent on parameter-passing mechanism
- Ada detects some cases and raises `Program_Error`

```
procedure Update (Doubled, Tripled : in out Integer);
...
Update (Doubled => A,
        Tripled => A);  -- illegal in Ada 2012
```

Functions' Parameter Modes

Ada 2012

- Can be mode `in out` and `out` too
- **Note:** operator functions can only have mode `in`
  - Including those you overload
  - Keeps readers sane
- Justification for only mode `in` prior to Ada 2012
  - No side effects: should be like mathematical functions
  - But side effects are still possible via globals
  - So worst possible case: side effects are possible and necessarily hidden!

Easy Cases Detected and Not Legal

```
procedure Example (A : in out Positive) is
   function Increment (This : Integer) return Integer is
   begin
      A := A + This;
      return A;
   end Increment;
   X : array (1 .. 10) of Integer;
begin
   -- order of evaluating A not specified
   X (A) := Increment (A);
end Example;
```

Extended Examples

Tic-Tac-Toe Winners Example (Spec)

```
package TicTacToe is
  type Players is (Nobody, X, O);
  type Move is range 1 .. 9;
  type Game is array (Move) of
    Players;
  function Winner (This : Game)
    return Players;
  ...
end TicTacToe;
```

1	N	2	N	3	N
4	N	5	N	6	N
7	N	8	N	9	N

Tic-Tac-Toe Winners Example (Body)

```
function Winner (This : Game) return Players is
  type Winning_Combinations is range 1 .. 8;
  type Required_Positions   is range 1 .. 3;
  Winning : constant array
    (Winning_Combinations, Required_Positions)
      of Move := (-- rows
                  (1, 2, 3), (4, 5, 6), (7, 8, 9),
                  -- columns
                  (1, 4, 7), (2, 5, 8), (3, 6, 9),
                  -- diagonals
                  (1, 5, 9), (3, 5, 7));

begin
  for K in Winning_Combinations loop
    if This (Winning (K, 1)) /= Nobody and then
      (This (Winning (K, 1)) = This (Winning (K, 2)) and
       This (Winning (K, 2)) = This (Winning (K, 3)))
    then
      return This (Winning (K, 1));
    end if;
  end loop;
  return Nobody;
end Winner;
```

Set Example

```
-- some colors
type Color is (Red, Orange, Yellow, Green, Blue, Violet);
-- truth table for each color
type Set is array (Color) of Boolean;
-- unconstrained array of colors
type Set_Literal is array (Positive range <>) of Color;

-- Take an array of colors and set table value to True
-- for each color in the array
function Make (Values : Set_Literal) return Set;
-- Take a color and return table with color value set to true
function Make (Base : Color) return Set;
-- Return True if the color has the truth value set
function Is_Member (C : Color; Of_Set: Set) return Boolean;

Null_Set : constant Set := (Set'Range => False);
RGB      : Set := Make (
             Set_Literal'(Red, Blue, Green));
Domain   : Set := Make (Green);

if Is_Member (Red, Of_Set => RGB) then ...

-- Type supports operations via Boolean operations,
-- as Set is a one-dimensional array of Boolean
S1, S2 : Set := Make (....);
Union : Set := S1 or S2;
Intersection : Set := S1 and S2;
Difference : Set := S1 xor S2;
```

Set Example (Implementation)

```
function Make (Base : Color) return Set is
  Result : Set := Null_Set;
begin
   Result (Base) := True;
   return Result;
end Make;

function Make (Values : Set_Literal) return Set is
  Result : Set := Null_Set;
begin
  for K in Values'Range loop
    Result (Values (K)) := True;
  end loop;
  return Result;
end Make;

function Is_Member (C: Color;
                    Of_Set: Set)
                    return Boolean is
begin
  return Of_Set(C);
end Is_Member;
```

Lab

Subprograms Lab

- Requirements
  - Build a list of sorted unique integers
    - Do not add an integer to the list if it is already there
  - Print the list
- Hints
  - Subprograms can be nested inside other subprograms
    - Like inside **main**
  - Build a Search subprogram to find the correct insertion point in the list

Subprograms Lab Solution - Search

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

function Search
  (List : List_T;
   Item : Integer)
   return Positive is
begin
   if List'Length = 0 then
      return 1;
   elsif Item <= List (List'First) then
      return 1;
   else
      for Idx in (List'First + 1) .. List'Length loop
         if Item <= List (Idx) then
            return Idx;
         end if;
      end loop;
      return List'Last;
   end if;
end Search;
```

Subprograms Lab Solution - Main

```
   procedure Add (Item : Integer) is
      Place : Natural := Search (List (1..Length), Item);
   begin
      if List (Place) /= Item then
         Length                      := Length + 1;
         List (Place + 1 .. Length) := List (Place .. Length - 1);
         List (Place)                := Item;
      end if;
   end Add;

begin

   Add (100);
   Add (50);
   Add (25);
   Add (50);
   Add (90);
   Add (45);
   Add (22);

   for Idx in 1 .. Length loop
      Put_Line (List (Idx)'Image);
   end loop;

end Main;
```

Summary

Summary

- `procedure` is abstraction for actions
- `function` is abstraction for value computations
- Separate declarations are sometimes necessary
  - Mutual recursion
  - Visibility from packages (i.e., exporting)
- Modes allow spec to define effects on actuals
  - Don't have to see the implementation: abstraction maintained
- Parameter-passing mechanism is based on the type
- Watch those side effects!

Type Derivation

Introduction

Type Derivation

- Type *derivation* allows for reusing code
- Type can be **derived** from a **base type**
- Base type can be substituted by the derived type
- Subprograms defined on the base type are **inherited** on derived type
- This is **not** OOP in Ada
  - Tagged derivation **is** OOP in Ada

Ada Mechanisms for Type Inheritance

- *Primitive* operations on types
  - Standard operations like + and -
  - Any operation that acts on the type
- Type derivation
  - Define types from other types that can add limitations
  - Can add operations to the type
- Tagged derivation
  - **This** is OOP in Ada
  - Seen in other chapter

Primitives

Primitive Operations

- A type is characterized by two elements
  - Its data structure
  - The set of operations that applies to it
- The operations are called **primitive operations** in Ada

```
type T is new Integer;
procedure Attrib_Function(Value : T);
```

General Rule For a Primitive

- Primitives are subprograms
- **S** is a primitive of type **T** iff
  - **S** is declared in the scope of **T**
  - **S** "uses" type **T**
    - As a parameter
    - As its return type (for `function`)
  - **S** is above *freeze-point*
- Rule of thumb
  - Primitives must be declared **right after** the type itself
  - In a scope, declare at most a **single** type with primitives

```
package P is
   type T is range 1 .. 10;
   procedure P1 (V : T);
   procedure P2 (V1 : Integer; V2 : T);
   function F return T;
end P;
```

Simple Derivation

Simple Type Derivation

- Any type (except `tagged`) can be derived

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

- Child inherits from:
  - The data **representation** of the parent
  - The **primitives** of the parent
- Conversions are possible from child to parent

```
type Parent is range 1 .. 10;
procedure Prim (V : Parent);
type Child is new Parent;  -- Freeze Parent
procedure Not_A_Primitive (V : Parent);
C : Child;
...
Prim (C);  -- Implicitly declared
Not_A_Primitive (Parent (C));
```

Simple Derivation and Type Structure

- The type "structure" can not change
  - `array` cannot become `record`
  - Integers cannot become floats
- But can be **constrained** further
- Scalar ranges can be reduced

```
type Tiny_Int is range -100 .. 100;
type Tiny_Positive is new Tiny_Int range 1 .. 100;
```

- Unconstrained types can be constrained

```
type Arr is array (Integer range <>) of Integer;
type Ten_Elem_Arr is new Arr (1 .. 10);
type Rec (Size : Integer) is record
   Elem : Arr (1 .. Size);
end record;
type Ten_Elem_Rec is new Rec (10);
```

Overriding Indications

Ada 2005

- **Optional** indications
- Checked by compiler

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

- **Replacing** a primitive: `overriding` indication

```
overriding procedure Prim (V : Child);
```

- **Adding** a primitive: `not overriding` indication

```
not overriding procedure Prim2 (V : Child);
```

- **Removing** a primitive: `overriding` as `abstract`

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

Quiz

```
type T1 is range 1 .. 100;
procedure Proc_A (X : in out T1);

type T2 is new T1 range 2 .. 99;
procedure Proc_B (X : in out T1);
procedure Proc_B (X : in out T2);

-- Other scope
procedure Proc_C (X : in out T2);

type T3 is new T2 range 3 .. 98;

procedure Proc_C (X : in out T3);
```

Which are T1's primitives

A. Proc_A
B. Proc_B
C. Proc_C
D. No primitives of T1

.

Quiz

```
type T1 is range 1 .. 100;
procedure Proc_A (X : in out T1);

type T2 is new T1 range 2 .. 99;
procedure Proc_B (X : in out T1);
procedure Proc_B (X : in out T2);

-- Other scope
procedure Proc_C (X : in out T2);

type T3 is new T2 range 3 .. 98;

procedure Proc_C (X : in out T3);
```

Which are T1's primitives

A. *Proc_A*
B. Proc_B
C. Proc_C
D. No primitives of T1

Explanations

A. Correct
B. Freeze: T1 has been derived
C. Freeze: scope change
D. Incorrect

.

Summary

Summary

- *Primitive* of a type
  - Subprogram above **freeze-point** that takes or return the type
  - Can be a primitive for **multiple types**
- Freeze point rules can be tricky
- Simple type derivation
  - Types derived from other types can only **add limitations**
    - Constraints, ranges
    - Cannot change underlying structure

Expressions

Introduction

Advanced Expressions

- Different categories of expressions above simple assignment and conditional statements
  - Constraining types to sub-ranges to increase readability and flexibility
    - Allows for simple membership checks of values
  - Embedded conditional assignments
    - Equivalent to C's `A ? B : C` and even more elaborate

Membership Tests

"Membership" Operation

- Syntax

```
simple_expression [not] in membership_choice_list
membership_choice_list ::= membership_choice
                           { | membership_choice}
membership_choice ::= expression | range | subtype_mark
```

- Acts like a boolean function
- Usable anywhere a boolean value is allowed

```
X : Integer := ...
B : Boolean := X in 0..5;
C : Boolean := X not in 0..5; -- also "not (X in 0..5)"
```

Testing Constraints via Membership

```
type Calendar_Days  is
    (Mon, Tues, Wed, Thur, Fri, Sat, Sun);
subtype Weekdays is Calendar_Days range Mon .. Fri;
Day : Calendar_Days := Today;
...
if Day in Mon .. Fri then ...
if Day in Weekdays then ... -- same as above
```

Testing Non-Contiguous Membership

Ada 2012

- Uses vertical bar "choice" syntax

```
declare
 M : Month_Number := Month (Clock);
begin
  if M in 9 | 4 | 6 | 11 then
    Put_Line ("31 days in this month");
  elsif M = 2 then
    Put_Line ("It's February, who knows?");
  else
    Put_Line ("30 days in this month");
  end if;
```

Quiz

```
type Days_T is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
subtype Weekdays_T is Days_T range Mon .. Fri;
Today : Days_T;
```

Which condition is **not** legal?

A. `if Today = Mon or Wed or Fri then`
B. `if Today in Days_T then`
C. `if Today not in Weekdays_T then`
D. `if Today in Tue | Thu then`

Quiz

```
type Days_T is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
subtype Weekdays_T is Days_T range Mon .. Fri;
Today : Days_T;
```

Which condition is **not** legal?

A. *`if Today = Mon or Wed or Fri then`*
B. `if Today in Days_T then`
C. `if Today not in Weekdays_T then`
D. `if Today in Tue | Thu then`

Explanations

A. To use `or`, both sides of the comparison must be duplicated (e.g. `Today = Mon or Today = Wed`)
B. Legal - should always return `True`
C. Legal - returns `True` if `Today` is `Sat` or `Sun`
D. Legal - returns `True` if `Today` is `Tue` or `Thu`

Qualified Names

Qualification

- Explicitly indicates the subtype of the value
- Syntax

```
qualified_expression ::= subtype_mark'(expression) |
                         subtype_mark'aggregate
```

- Similar to conversion syntax
  - Mnemonic - "qualification uses quote"
- Various uses shown in course
  - Testing constraints
  - Removing ambiguity of overloading
  - Enhancing readability via explicitness

Testing Constraints via Qualification

- Asserts value is compatible with subtype
  - Raises exception `Constraint_Error` if not true

```
subtype Weekdays is Days range Mon .. Fri;
This_Day : Days;
...
case Weekdays'(This_Day) is --runtime error if out of range
  when Mon =>
    Arrive_Late;
    Leave_Early;
  when Tue .. Thur =>
    Arrive_Early;
    Leave_Late;
  when Fri =>
    Arrive_Early;
    Leave_Early;
end case; -- no 'others' because all subtype values covered
```

Conditional Expressions

Conditional Expressions

Ada 2012

- Ultimate value depends on a controlling condition
- Allowed wherever an expression is allowed
  - Assignment RHS, formal parameters, aggregates, etc.
- Similar intent as in other languages
  - Java, C/C++ ternary operation **A ? B : C**
  - Python conditional expressions
  - etc.
- Two forms:
  - *If expressions*
  - *Case expressions*

If Expressions

Ada 2012

- Syntax looks like an if-statement without **end if**

```
if_expression ::=
   (if condition then dependent_expression
   {elsif condition then dependent_expression}
   [else dependent_expression])
condition ::= boolean_expression
```

  - The conditions are always Boolean values

```
(if Today > Wednesday then 1 else 0)
```

Result Must Be Compatible with Context

- The **dependent_expression** parts, specifically

```
X : Integer :=
    (if Day_Of_Week (Clock) > Wednesday then 1 else 0);
```

If Expression Example

```
declare
  Remaining : Natural := 5;  -- arbitrary
begin
  while Remaining > 0 loop
    Put_Line ("Warning! Self-destruct in" &
      Remaining'Image &
      (if Remaining = 1 then " second" else " seconds"));
    delay 1.0;
    Remaining := Remaining - 1;
  end loop;
  Put_Line ("Boom! (goodbye Nostromo)");
```

Boolean If-Expressions

- Return a value of either True or False
  - `(if P then Q)` - assuming **P** and **Q** are **Boolean**
  - "If P is True then the result of the if-expression is the value of Q"
- But what is the overall result if all conditions are False?
- Answer: the default result value is True
  - Why?
    - Consistency with mathematical proving

The **else** Part When Result Is Boolean

- Redundant because the default result is True

```
(if P then Q else True)
```

- So for convenience and elegance it can be omitted

```
Acceptable : Boolean := (if P1 > 0 then P2 > 0 else True);
Acceptable : Boolean := (if P1 > 0 then P2 > 0);
```

- Use `else` if you need to return False at the end

Rationale for Parentheses Requirement

- Prevents ambiguity regarding any enclosing expression
- Problem:

```
X : Integer := if condition then A else B + 1;
```

- Does that mean
  - If condition, then **X := A + 1**, else **X := B + 1 OR**
  - If condition, then **X := A**, else **X := B + 1**
- But not required if parentheses already present
  - Because enclosing construct includes them

```
Subprogram_Call(if A then B else C);
```

When To Use *If Expressions*

- When you need computation to be done prior to sequence of statements
  - Allows constants that would otherwise have to be variables
- When an enclosing function would be either heavy or redundant with enclosing context
  - You'd already have written a function if you'd wanted one
- Preconditions and postconditions
  - All the above reasons
  - Puts meaning close to use rather than in package body
- Static named numbers
  - Can be much cleaner than using Boolean'Pos(condition)

If Expression Example for Constants

- Starting from

```
End_of_Month : array (Months) of Days
  := (Sep | Apr | Jun | Nov => 30,
     Feb => 28,
     others => 31);
begin
  if Leap (Today.Year) then -- adjust for leap year
    End_of_Month (Feb) := 29;
  end if;
  if Today.Day = End_of_Month(Today.Month) then
...
```

- Using if-expression to call `Leap (Year)` as needed

```
End_Of_Month : constant array (Months) of Days
  := (Sep | Apr | Jun | Nov => 30,
      Feb => (if Leap (Today.Year)
              then 29 else 28),
      others => 31);
begin
  if Today.Day /= End_of_Month(Today.Month) then
...
```

Case Expressions

Ada 2012

- Syntax similar to `case` statements
  - Lighter: no closing **end case**
  - Commas between choices
- Same general rules as *if expressions*
  - Parentheses required unless already present
  - Type of "result" must match context
- Advantage over *if expressions* is completeness checked by compiler
- Same as with `case` statements (unless `others` is used)

```
-- compile error if not all days covered
Hours : constant Integer :=
   (case Day_of_Week is
    when Mon .. Thurs => 9,
    when Fri          => 4,
    when Sat | Sun    => 0);
```

Case Expression Example

```
Leap : constant Boolean :=
   (Today.Year mod 4 = 0 and Today.Year mod 100 /= 0)
   or else
   (Today.Year mod 400 = 0);
End_Of_Month : array (Months) of Days;
...
-- initialize array
for M in Months loop
  End_Of_Month (M) :=
     (case M is
      when Sep | Apr | Jun | Nov => 30,
      when Feb => (if Leap then 29 else 28),
      when others => 31);
end loop;
```

Quiz

```
function Sqrt (X : Float) return Float;
F : Float;
B : Boolean;
```

Which statement is **not** legal?

A. `F := if X < 0.0 then Sqrt (-1.0 * X) else Sqrt (X);`
B. `F := Sqrt(if X < 0.0 then -1.0 * X else X);`
C. `B := (if X < 0.0 then Sqrt (-1.0 * X) < 10.0 else True);`
D. `B := (if X < 0.0 then Sqrt (-1.0 * X) < 10.0);`

Quiz

```
function Sqrt (X : Float) return Float;
F : Float;
B : Boolean;
```

Which statement is **not** legal?

A. *F := if X < 0.0 then Sqrt (-1.0 * X) else Sqrt (X);*
B. `F := Sqrt(if X < 0.0 then -1.0 * X else X);`
C. `B := (if X < 0.0 then Sqrt (-1.0 * X) < 10.0 else True);`
D. `B := (if X < 0.0 then Sqrt (-1.0 * X) < 10.0);`

Explanations

A. Missing parentheses around expression
B. Legal - Expression is already enclosed in parentheses so you don't need to add more
C. Legal - **else** True not needed but is allowed
D. Legal - B will be True if X >= 0.0

Lab

Expressions Lab

- Requirements
  - Allow the user to fill a list with dates
  - After the list is created, create functions to print True/False if ...
    - Any date is not legal (taking into account leap years!)
    - All dates are in the same calendar year
  - Use *expression functions* for all validation routines
- Hints
  - Use subtype membership for range validation
  - You will need *conditional expressions* in your functions
  - You *can* use component-based iterations for some checks
    - But you *must* use indexed-based iterations for others

Expressions Lab Solution - Checks

```
subtype Year_T is Positive range 1_900 .. 2_099;
subtype Month_T is Positive range 1 .. 12;
subtype Day_T is Positive range 1 .. 31;

type Date_T is record
   Year  : Positive;
   Month : Positive;
   Day   : Positive;
end record;

List : array (1 .. 5) of Date_T;
Item : Date_T;

function Is_Leap_Year (Year : Positive)
                       return Boolean is
  (Year mod 400 = 0 or else (Year mod 4 = 0 and Year mod 100 /= 0));

function Days_In_Month (Month : Positive;
                        Year  : Positive)
                        return Day_T is
  (case Month is when 4 | 6 | 9 | 11 => 30,
     when 2 => (if Is_Leap_Year (Year) then 29 else 28), when others => 31);

function Is_Valid (Date : Date_T)
                   return Boolean is
  (Date.Year in Year_T and then Date.Month in Month_T
   and then Date.Day <= Days_In_Month (Date.Month, Date.Year));

function Any_Invalid return Boolean is
begin
   for Date of List loop
      if not Is_Valid (Date) then
         return True;
      end if;
   end loop;
   return False;
end Any_Invalid;

function Same_Year return Boolean is
begin
   for Index in List'range loop
      if List (Index).Year /= List (List'first).Year then
         return False;
      end if;
   end loop;
   return True;
end Same_Year;
```

Expressions Lab Solution - Main

```
   function Number (Prompt : String)
                    return Positive is
   begin
      Put (Prompt & "> ");
      return Positive'Value (Get_Line);
   end Number;

begin

   for I in List'Range loop
      Item.Year  := Number ("Year");
      Item.Month := Number ("Month");
      Item.Day   := Number ("Day");
      List (I)   := Item;
   end loop;

   Put_Line ("Any invalid: " & Boolean'image (Any_Invalid));
   Put_Line ("Same Year: " & Boolean'image (Same_Year));

end Main;
```

Summary

Summary

- Conditional expressions are allowed wherever expressions are allowed, but beware over-use
    - Especially useful when a constant is intended
    - Especially useful when a static expression is required

Quantified Expressions

Quantified Expressions

Introduction

Ada 2012

- Expressions that have a Boolean value
- The value indicates something about a set of objects
  - In particular, whether something is True about that set
- That "something" is expressed as an arbitrary boolean expression
  - A so-called "predicate"
- "Universal" quantified expressions
  - Indicate whether predicate holds for all components
- "Existential" quantified expressions
  - Indicate whether predicate holds for at least one component

Examples

```
with GNAT.Random_Numbers; use GNAT.Random_Numbers;
with Ada.Text_IO;         use Ada.Text_IO;
procedure Quantified_Expressions is
   Gen    : Generator;
   Values : constant array (1 .. 10) of Integer := (others => Random (Gen));

   Any_Even : constant Boolean := (for some N of Values => N mod 2 = 0);
   All_Odd  : constant Boolean := (for all N of reverse Values => N mod 2 = 1);

   function Is_Sorted return Boolean is
     (for all K in Values'Range =>
        K = Values'First or else Values (K - 1) <= Values (K));

   function Duplicate return Boolean is
     (for some I in Values'Range =>
        (for some J in I + 1 .. Values'Last => Values (I) = Values (J)));

begin
   Put_Line ("Any Even: " & Boolean'Image (Any_Even));
   Put_Line ("All Odd: " & Boolean'Image (All_Odd));
   Put_Line ("Is_Sorted " & Boolean'Image (Is_Sorted));
   Put_Line ("Duplicate " & Boolean'Image (Duplicate));
end Quantified_Expressions;
```

Semantics Are As If You Wrote This Code

Ada 2012

```
function Universal (Set : Components) return Boolean is
begin
  for C of Set loop
    if not Predicate (C) then
      return False;  -- Predicate must be true for all
    end if;
  end loop;
  return True;
end Universal;

function Existential (Set : Components) return Boolean is
begin
  for C of Set loop
    if Predicate (C) then
      return True;  -- Predicate need only be true for one
    end if;
  end loop;
  return False;
end Existential;
```

Quantified Expressions Syntax

Ada 2012

- Four `for` variants
  - Index-based `in` or component-based `of`
  - Existential `some` or universal `all`
- Using arrow => to indicate *predicate* expression

```
(for some Index in Subtype_T => Predicate (Index))
(for all Index in Subtype_T => Predicate (Index))
(for some Value of Container_Obj => Predicate (Value))
(for all Value of Container_Obj => Predicate (Value))
```

Simple Examples

Ada 2012

```
Values : constant array (1 .. 10) of Integer := (...);
Is_Any_Even : constant Boolean :=
   (for some V of Values => V mod 2 = 0);
Are_All_Even : constant Boolean :=
   (for all V of Values => V mod 2 = 0);
```

Universal Quantifier

Ada 2012

- In logic, denoted by ∀ (inverted 'A', for "all")
- "There is no member of the set for which the predicate does not hold"
  - If predicate is False for any member, the whole is False
- Functional equivalent

```
function Universal (Set : Components) return Boolean is
begin
  for C of Set loop
    if not Predicate (C) then
       return False; -- Predicate must be true for all
    end if;
  end loop;
  return True;
end Universal;
```

Universal Quantifier Illustration

Ada 2012

- "There is no member of the set for which the predicate does not hold"
- Given a set of integer answers to a quiz, there are no answers that are not 42 (i.e., all are 42)

```
Ultimate_Answer : constant := 42; -- to everything...
Answers : constant array (1 .. 10)
    of Integer := (...);
All_Correct_1 : constant Boolean :=
   (for all Component of Answers =>
      Component = Ultimate_Answer);
All_Correct_2 : constant Boolean :=
   (for all K in Answers'range =>
      Answers(K) = Ultimate_Answer);
```

Universal Quantifier Real-World Example

Ada 2012

```
type DMA_Status_Flag is (...);
function Status_Indicated (
  Flag : DMA_Status_Flag)
  return Boolean;
None_Set : constant Boolean := (
  for all Flag in DMA_Status_Flag =>
    not Status_Indicated (Flag));
```

Existential Quantifier

Ada 2012

- In logic, denoted by ∃ (rotated 'E', for "exists")
- "There is at least one member of the set for which the predicate holds"
  - If predicate is True for any member, the whole is True
- Functional equivalent

```
function Existential (Set : Components) return Boolean is
begin
  for C of Set loop
    if Predicate (C) then
      return True; -- Need only be true for at least one
    end if;
  end loop;
  return False;
end Existential;
```

Existential Quantifier Illustration

Ada 2012

- "There is at least one member of the set for which the predicate holds"
- Given set of Integer answers to a quiz, there is at least one answer that is 42

```
Ultimate_Answer : constant := 42; -- to everything...
Answers : constant array (1 .. 10)
    of Integer := (...);
Any_Correct_1 : constant Boolean :=
   (for some Component of Answers =>
      Component = Ultimate_Answer);
Any_Correct_2 : constant Boolean :=
   (for some K in Answers'range =>
      Answers(K) = Ultimate_Answer);
```

Index-Based vs Component-Based Indexing

Ada 2012

- Given an array of Integers

```
Values : constant array (1 .. 10) of Integer := (...);
```

- Component-based indexing is useful for checking individual values

```
Contains_Negative_Number : constant Boolean :=
   (for some N of Values => N < 0);
```

- Index-based indexing is useful for comparing across values

```
Is_Sorted : constant Boolean :=
  (for all I in Values'Range =>
    I = Values'first or else Values(I) >= Values(I-1));
```

"Pop Quiz" for Quantified Expressions

Ada 2012

- What will be the value of **Ascending_Order**?

```
Table : constant array (1 .. 10) of Integer :=
      (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
Ascending_Order : constant Boolean := (
  for all K in Table'Range =>
    K > Table'First and then Table (K - 1) <= Table (K));
```

  - Answer: **False**. Predicate fails when **K = Table'First**
    - First subcondition is False!
    - Condition should be

```
Ascending_Order : constant Boolean := (
   for all K in Table'Range =>
      K = Table'first or else Table (K - 1) <= Table (K));
```

When The Set Is Empty...

Ada 2012

- Universally quantified expressions are True
  - Definition: there is no member of the set for which the predicate does not hold
  - If the set is empty, there is no such member, so True
  - "All people 12-feet tall will be given free chocolate."
- Existentially quantified expressions are False
  - Definition: there is at least one member of the set for which the predicate holds
- If the set is empty, there is no such member, so False
- Common convention in set theory, arbitrary but settled

Not Just Arrays: Any "Iterable" Objects

Ada 2012

- Those that can be iterated over
- Language-defined, such as the containers
- User-defined too

```
package Characters is new
   Ada.Containers.Vectors (Positive, Character);
use Characters;
Alphabet  : constant Vector := To_Vector('A',1) & 'B' & 'C';
Any_Zed   : constant Boolean :=
           (for some C of Alphabet => C = 'Z');
All_Lower : constant Boolean :=
            (for all C of Alphabet => Is_Lower (C));
```

Conditional / Quantified Expression Usage

Ada 2012

- Use them when a function would be too heavy
- Don't over-use them!

```
if (for some Component of Answers =>
    Component = Ultimate_Answer)
then
```

- Function names enhance readability
    - So put the quantified expression in a function

    ```
    if At_Least_One_Answered (Answers) then
    ```

- Even in pre/postconditions, use functions containing quantified expressions for abstraction

Quiz

Which declaration(s) is(are) legal?

A.
```
function F (S : String) return Boolean is
  (for all C of S => C /= ' ');
```

B.
```
function F (S : String) return Boolean is
  (not for some C of S => C = ' ');
```

C.
```
function F (S : String) return String is
  (for all C of S => C);
```

D.
```
function F (S : String) return String is
  (if (for all C of S => C /= ' ') then "OK"
   else "NOK");
```

Quiz

Which declaration(s) is(are) legal?

A. *function F (S : String) return Boolean is (for all C of S => C /= ' ');*

```
function F (S : String) return Boolean is
  (for all C of S => C /= ' ');
```

B.

```
function F (S : String) return Boolean is
  (not for some C of S => C = ' ');
```

C.

```
function F (S : String) return String is
  (for all C of S => C);
```

D. *function F (S : String) return String is (if (for all C of S => C /= ' ') then "OK" else "NOK");*

```
function F (S : String) return String is
  (if (for all C of S => C /= ' ') then "OK"
   else "NOK");
```

B. Parentheses required around the quantified expression
C. Must return a `Boolean`

Quiz

```ada
type T1 is array (1 .. 3) of Integer;
type T2 is array (1 .. 3) of Integer;
```

Which piece(s) of code correctly perform(s) equality check on A and B?

A.
```ada
function "=" (A : T1; B : T2) return Boolean is
  (A = T1 (B));
```

B.
```ada
function "=" (A : T1; B : T2) return Boolean is
  (for all E1 of A => (for all E2 of B => E1 = E2));
```

C.
```ada
function "=" (A : T1; B : T2) return Boolean is
  (for some E1 of A => (for some E2 of B => E1 =
E2));
```

D.
```ada
function "=" (A : T1; B : T2) return Boolean is
  (for all J in A'Range => A (J) = B (J));
```

Quiz

```
type T1 is array (1 .. 3) of Integer;
type T2 is array (1 .. 3) of Integer;
```

Which piece(s) of code correctly perform(s) equality check on A and B?

A. *`function "=" (A : T1; B : T2) return Boolean is (A = T1 (B));`*

B. `function "=" (A : T1; B : T2) return Boolean is (for all E1 of A => (for all E2 of B => E1 = E2));`

C. `function "=" (A : T1; B : T2) return Boolean is (for some E1 of A => (for some E2 of B => E1 = E2));`

D. *`function "=" (A : T1; B : T2) return Boolean is (for all J in A'Range => A (J) = B (J));`*

B. Counterexample: `A = B = (0, 1, 0)` returns `False`

C. Counterexample: `A = (0, 0, 1)` and `B = (0, 1, 1)` returns `True`

Quiz

```
type Array1_T is array (1 .. 3) of Integer;
type Array2_T is array (1 .. 3) of Array1_T;
A : Array2_T;
```

The above describes an array A whose elements are arrays of three elements. Which expression would one use to determine if at least one of A's elements are sorted?

A. `(for some El of A => (for some Idx in 2 .. 3 =>
      El (Idx) >= El (Idx - 1)));`

B. `(for all El of A => for all Idx in 2 .. 3 =>
      El (Idx) >= El (Idx - 1)));`

C. `(for some El of A => (for all Idx in 2 .. 3 =>
      El (Idx) >= El (Idx - 1)));`

D. `(for all El of A => (for some Idx in 2 .. 3 =>
      El (Idx) >= El (Idx - 1)));`

Quiz

```
type Array1_T is array (1 .. 3) of Integer;
type Array2_T is array (1 .. 3) of Array1_T;
A : Array2_T;
```

The above describes an array A whose elements are arrays of three elements. Which expression would one use to determine if at least one of A's elements are sorted?

A.
```
(for some El of A => (for some Idx in 2 .. 3 =>
   El (Idx) >= El (Idx - 1)));
```

B.
```
(for all El of A => for all Idx in 2 .. 3 =>
   El (Idx) >= El (Idx - 1)));
```

C.
```
(for some El of A => (for all Idx in 2 .. 3 =>
   El (Idx) >= El (Idx - 1)));
```

D.
```
(for all El of A => (for some Idx in 2 .. 3 =>
    El (Idx) >= El (Idx - 1)));
```

A. Will be `True` if any element has two consecutive increasing values
B. Will be `True` if every element is sorted
C. Correct
D. Will be `True` if every element has two consecutive increasing values

Lab

Advanced Expressions Lab

- Requirements
  - Allow the user to fill a list with dates
  - After the list is created, use *quantified expressions* to print True/False
    - If any date is not legal (taking into account leap years!)
    - If all dates are in the same calendar year
  - Use *expression functions* for all validation routines
- Hints
  - Use subtype membership for range validation
  - You will need *conditional expressions* in your functions
  - You *can* use component-based iterations for some checks
    - But you *must* use indexed-based iterations for others
  - This is the same lab as the *Expressions* lab, we're just replacing the validation functions with quantified expressions!
    - So you can just copy that project and update the code!

Advanced Expressions Lab Solution - Checks

```
subtype Year_T is Positive range 1_900 .. 2_099;
subtype Month_T is Positive range 1 .. 12;
subtype Day_T is Positive range 1 .. 31;

type Date_T is record
   Year  : Positive;
   Month : Positive;
   Day   : Positive;
end record;

List : array (1 .. 5) of Date_T;
Item : Date_T;

function Is_Leap_Year (Year : Positive)
                       return Boolean is
  (Year mod 400 = 0 or else (Year mod 4 = 0 and Year mod 100 /= 0));

function Days_In_Month (Month : Positive;
                        Year  : Positive)
                        return Day_T is
  (case Month is when 4 | 6 | 9 | 11 => 30,
     when 2 => (if Is_Leap_Year (Year) then 29 else 28), when others => 31);

function Is_Valid (Date : Date_T)
                   return Boolean is
  (Date.Year in Year_T and then Date.Month in Month_T
   and then Date.Day <= Days_In_Month (Date.Month, Date.Year));

function Any_Invalid return Boolean is
  (for some Date of List => not Is_Valid (Date));

function Same_Year return Boolean is
  (for all I in List'range => List (I).Year = List (List'first).Year);
```

Advanced Expressions Lab Solution - Main

```
   function Number (Prompt : String)
                    return Positive is
   begin
      Put (Prompt & "> ");
      return Positive'Value (Get_Line);
   end Number;

begin

   for I in List'Range loop
      Item.Year  := Number ("Year");
      Item.Month := Number ("Month");
      Item.Day   := Number ("Day");
      List (I)   := Item;
   end loop;

   Put_Line ("Any invalid: " & Boolean'image (Any_Invalid));
   Put_Line ("Same Year: " & Boolean'image (Same_Year));

end Main;
```

Summary

Summary

- Quantified expressions are general purpose but especially useful with pre/postconditions
    - Consider hiding them behind expressive function names

Overloading

Introduction

Introduction

- *Overloading* is the use of an already existing name to define a **new** entity
- Historically, only done as part of the language **implementation**
  - Eg. on operators
  - Float vs Integer vs pointers arithmetic
- Several languages allow **user-defined** overloading
  - C++
  - Python (limited to operators)
  - Haskell

Visibility and Scope

- Overloading is **not** re-declaration
- Both entities **share** the name
  - No hiding
  - Compiler performs **name resolution**
- Allowed to be declared in the **same scope**
  - Remember this is forbidden for "usual" declarations

Overloadable Entities In Ada

- Identifiers for subprograms
  - Both procedure and function names
- Identifiers for enumeration values (enumerals)
- Language-defined operators for functions

```
procedure Put (Str : in String);
procedure Put (C : in Complex);
function Max (Left, Right : Integer) return Integer;
function Max (Left, Right : Float)   return Float;
function "+" (Left, Right : Rational) return Rational;
function "+" (Left, Right : Complex)  return Complex;
function "*" (Left : Natural; Right : Character)
      return String;
```

Function Operator Overloading Example

```
--  User-defined overloading
function "+" (L,R : Complex) return Complex is
begin
  return (L.Real_Part + R.Real_Part,
          L.Imaginary + R.Imaginary);
end "+";

A, B, C : Complex;
I, J, K : Integer;

I := J + K; -- overloaded operator (predefined)
A := B + C; -- overloaded operator (user-defined)
```

Benefits and Risk of Overloading

- Management of the name space
  - Support for abstraction
  - Linker will not simply take the first match and apply it globally
- Safe: compiler will reject ambiguous calls
- Sensible names are the programmer's job

```
function "+" (L, R : Integer) return String is
begin
   return Integer'Image (L - R);
end "+";
```

Enumerals and Operators

Overloading Enumerals

- Each is treated as if a function name (identifier)
- Thus same rules as for function identifier overloading

```
type Stop_Light is (Red, Yellow, Green);
type Colors is (Red, Blue, Green);
Shade : Colors := Red;
Current_Value : Stop_Light := Red;
```

Overloadable Operator Symbols

- Only those defined by the language already
  - Users cannot introduce new operator symbols
- Note that assignment (:=) is not an operator
- Operators (in precedence order)

Logicals	and, or, xor
Relationals	`<`, `<=`, `=`, `>=`, `>`
Unary	`+`, `-`
Binary	`+`, `-`, `&`
Multiplying	`*`, `/`, `mod`, `rem`
Highest precedence	`**`, `abs`, `not`

Parameters for Overloaded Operators

- Must not change syntax of calls
  - Number of parameters must remain same (unary, binary...)
  - No default expressions allowed for operators
- Infix calls use positional parameter associations
  - Left actual goes to first formal, right actual goes to second formal
  - Definition

    ```
    function "*" (Left, Right : Integer) return Integer;
    ```

  - Usage

    ```
    X := 2 * 3;
    ```

- Named parameter associations allowed but ugly
  - Requires prefix notation for call

```
X := "*" (Left => 2, Right => 3);
```

Call Resolution

Call Resolution

- Compilers must reject ambiguous calls
- *Resolution* is based on the calling context
  - Compiler attempts to find a matching **profile**
  - Based on **Parameter** and **Result** Type
- Overloading is not re-definition, or hiding
  - More than one matching profile is ambiguous

```
type Complex is ...
function "+" (L, R : Complex) return Complex;
A, B : Complex := some_value;
C : Complex := A + B;
D : Float := A + B;  -- illegal!
E : Float := 1.0 + 2.0;
```

Profile Components Used

- Significant components appear in the call itself
  - **Number** of parameters
  - **Order** of parameters
  - **Base type** of parameters
  - **Result** type (for functions)
- Insignificant components might not appear at call
  - Formal parameter **names** are optional
  - Formal parameter **modes** never appear
  - Formal parameter **subtypes** never appear
  - **Default** expressions never appear

```
Display (X);
Display (Foo => X);
Display (Foo => X, Bar => Y);
```

Manually Disambiguating Calls

- Qualification can be used
- Named parameter association can be used
  - Unless name is ambiguous

```
type Stop_Light is (Red, Yellow, Green);
type Colors is (Red, Blue, Green);
procedure Put (Light : in Stop_Light);
procedure Put (Shade : in Colors);

Put (Red);  -- ambiguous call
Put (Yellow);  -- not ambiguous: only 1 Yellow
Put (Colors'(Red)); -- using type to distinguish
Put (Light => Green); -- using profile to distinguish
```

Overloading Example

```
function "+" (Left : Position; Right : Offset)
  return Position is
begin
   return Position'(Left.Row + Right.Row, Left.Column + Right.Col);
end "+";

function Acceptable (P : Position) return Boolean;
type Positions is array (Moves range <>) of Position;

function Next (Current : Position) return Positions is
  Result : Positions (Moves range 1 .. 4);
  Count  : Moves := 0;
  Test   : Position;
begin
  for K in Offsets'Range loop
    Test := Current + Offsets(K);
    if Acceptable (Test) then
      Count := Count + 1;
      Result (Count) := Test;
    end if;
  end loop;
  return Result (1 .. Count);
end Next;
```

Quiz

```
type Vertical_T is (Top, Middle, Bottom);
type Horizontal_T is (Left, Middle, Right);
function "*" (H : Horizontal_T; V : Vertical_T) return Positive;
function "*" (V : Vertical_T; H : Horizontal_T) return Positive;
P : Positive;
```

Which statement is not legal?

A. `P := Horizontal_T'(Middle) * Middle;`
B. `P := Top * Right;`
C. `P := "*" (Middle, Top);`
D. `P := "*" (H => Middle, V => Top);`

Quiz

```
type Vertical_T is (Top, Middle, Bottom);
type Horizontal_T is (Left, Middle, Right);
function "*" (H : Horizontal_T; V : Vertical_T) return Positive;
function "*" (V : Vertical_T; H : Horizontal_T) return Positive;
P : Positive;
```

Which statement is not legal?

A. `P := Horizontal_T'(Middle) * Middle;`
B. `P := Top * Right;`
C. `P := "*" (Middle, Top);`
D. *`P := "*" (H => Middle, V => Top);`*

Explanations

A. Qualifying one parameter resolves ambiguity
B. No overloaded names
C. Use of `Top` resolves ambiguity
D. When overloading subprogram names, best to not just switch the order of parameters

User-Defined Equality

User-Defined Equality

- Allowed like any other operator
  - Must remain a binary operator
- Typically declared as `return Boolean`
- Hard to do correctly for composed types
  - Especially **user-defined** types
  - Issue of *Composition of equality*

Lab

Overloading Lab

- Requirements
  - Create multiple functions named "Convert" to convert between digits and text representation
    - One routine should take a digit and return the text version (e.g. **3** would return **three**)
    - One routine should take text and return the digit (e.g. **two** would return **2**)
  - Query the user to enter text or a digit and print it's equivalent
  - If the user enters consecutive entries that are equivalent, print a message
    - e.g. **4** followed by **four** should get the message
- Hints
  - You can use enumerals for the text representation
    - Then use *'image* / *'value* where needed
  - Use an equivalence function two compare different types

Overloading Lab Solution - Conversion Functions

```
type Digit_T is range 0 .. 9;
type Digit_Name_T is
  (Zero, One, Two, Three, Four, Five, Six, Seven, Eight, Nine);

function Convert (Value : Digit_T) return Digit_Name_T;
function Convert (Value : Digit_Name_T) return Digit_T;
function Convert (Value : Character) return Digit_Name_T;
function Convert (Value : String) return Digit_T;

function "=" (L : Digit_Name_T; R : Digit_T) return Boolean is (Convert (L) = R);

function Convert (Value : Digit_T) return Digit_Name_T is
  (case Value is when 0 => Zero,  when 1 => One,
                 when 2 => Two,   when 3 => Three,
                 when 4 => Four,  when 5 => Five,
                 when 6 => Six,   when 7 => Seven,
                 when 8 => Eight, when 9 => Nine);

function Convert (Value : Digit_Name_T) return Digit_T is
  (case Value is when Zero  => 0, when One => 1,
                 when Two   => 2, when Three => 3,
                 when Four  => 4, when Five => 5,
                 when Six   => 6, when Seven => 7,
                 when Eight => 8, when Nine => 9);

function Convert (Value : Character) return Digit_Name_T is
  (case Value is when '0' => Zero,  when '1' => One,
                 when '2' => Two,   when '3' => Three,
                 when '4' => Four,  when '5' => Five,
                 when '6' => Six,   when '7' => Seven,
                 when '8' => Eight, when '9' => Nine,
                 when others => Zero);

function Convert (Value : String) return Digit_T is
  (Convert (Digit_Name_T'Value (Value)));
```

Overloading Lab Solution - Main

```
   Last_Entry : Digit_T := 0;

begin
   loop
      Put ("Input: ");
      declare
         Str : constant String := Get_Line;
      begin
         exit when Str'Length = 0;
         if Str (Str'First) in '0' .. '9' then
            declare
               Converted : constant Digit_Name_T := Convert (Str (Str'First));
            begin
               Put (Digit_Name_T'Image (Converted));
               if Converted = Last_Entry then
                  Put_Line (" - same as previous");
               else
                  Last_Entry := Convert (Converted);
                  New_Line;
               end if;
            end;
         else
            declare
               Converted : constant Digit_T := Convert (Str);
            begin
               Put (Digit_T'Image (Converted));
               if Converted = Last_Entry then
                  Put_Line (" - same as previous");
               else
                  Last_Entry := Converted;
                  New_Line;
               end if;
            end;
         end if;
      end;
   end loop;
end Main;
```

Summary

Summary

- Ada allows user-defined overloading
  - Identifiers and operator symbols
- Benefits easily outweigh danger of senseless names
  - Can have nonsensical names without overloading
- Compiler rejects ambiguous calls
- Resolution is based on the calling context
  - *Parameter and Result Type Profile*
- Calling context is those items present at point of call
  - Thus modes etc. don't affect overload resolution
- User-defined equality is allowed
  - But is tricky

Library Units

Introduction

Modularity

- Ability to split large system into subsystems
- Each subsystem can have its own components
- And so on ...

Library Units

Library Units

- Those not nested within another program unit
- Candidates
  - Subprograms
  - Packages
  - Generic Units
  - Generic Instantiations
  - Renamings
- Restrictions
  - No library level tasks
    - They are always nested within another unit
  - No overloading at library level
  - No library level functions named as operators

Library Units

```
package Operating_System is
  procedure Foo(...);
  procedure Bar(...);
  package Process_Manipulation is
    ...
  end Process_Manipulation;
  package File_System is
    ...
  end File_System;
end Operating_System;
```

- **Operating_System** is library unit
- **Foo**, **Bar**, etc - not library units

No 'Object' Library Items

```
package Library_Package is
   ...
end Library_Package;

-- Illegal: no such thing as "file scope"
Library_Object : Integer;

procedure Library_Procedure;

function Library_Function (Formal : in out Integer) is
  Local : Integer;
begin
   ...
end Library_Function;
```

Declared Object "Lifetimes"

- Same as their enclosing declarative region
  - Objects are always declared within some declarative region
- No `static` etc. directives as in C
- Objects declared within any subprogram
  - Exist only while subprogram executes

```
procedure Library_Subprogram is
  X : Integer;
  Y : Float;
begin
  ...
end Library_Subprogram;
```

Objects In Library Packages

- Exist as long as program executes (i.e., "forever")

```
package Named_Common is
  X : Integer; -- valid object for life of application
  Y : Float;    -- valid object for life of application
end Named_Common;
```

Objects In Non-library Packages

- Exist as long as region enclosing the package

```
procedure P is
  X : Integer; -- available while in P and Inner
  package Inner is
    Z : Boolean; -- available while in Inner
  end Inner;
  Y : Float; -- available while in P
begin
  ...
end P;
```

Program "Lifetime"

- Run-time library is initialized
- All (any) library packages are elaborated
  - Declarations in package declarative part are elaborated
  - Declarations in package body declarative part are elaborated
  - Executable part of package body is executed (if present)
- Main program's declarative part is elaborated
- Main program's sequence of statements executes
- Program executes until all threads terminate
- All objects in library packages cease to exist
- Run-time library shuts down

Library Unit Subprograms

- Recall: separate declarations are optional
  - Body can act as declaration if no declaration provided
- Separate declaration provides usual benefits
  - Changes/recompilation to body only require relinking clients
- File 1 (p.ads for GNAT)

```
procedure P (F : in Integer);
```

- File 2 (p.adb for GNAT)

```
procedure P (F : in Integer) is
begin
   ...
end P;
```

Library Unit Subprograms

- Specifications in declaration and body must conform
  - Example
    - Spec for P

```
procedure P (F : in Integer);
```

    - Body for P

```
procedure P (F : in float) is
begin
...
end P;
```

  - Declaration creates subprogram **P** in library
  - Declaration exists so body does not act as declaration
  - Compilation of file "p.adb" must fail
- New declaration with same name replaces old one
- Thus cannot overload library units

Main Subprograms

- Must be library subprograms
- No special program unit name required
- Can be many per program library
- Always can be procedures
- Can be functions if implementation allows it
    - Execution environment must know how to handle result

```
with Ada.Text_IO;
procedure Hello is
begin
  Ada.Text_IO.Put("Hello World");
end Hello;
```

Dependencies

with Clauses

- Specify the library units that a compilation unit depends upon
    - The "context" in which the unit is compiled
- Syntax (simplified)

```
context_clause ::= { context_item }
context_item ::= with_clause | use_clause
with_clause ::= with library_unit_name
                { , library_unit_name };
```

```
with Ada.Text_IO; -- dependency
procedure Hello is
begin
  Ada.Text_IO.Put ("Hello World");
end Hello;
```

with Clauses Syntax

- Helps explain restrictions on library units
  - No overloaded library units
  - If overloading allowed, which **P** would `with P;` refer to?
  - No library unit functions names as operators
    - Mostly because of no overloading

What To Import

- Need only name direct dependencies
  - Those actually referenced in the corresponding unit
- Will not cause compilation of referenced units
  - Unlike "include directives" of some languages

```
package A is
  type Something is ...
end A;

with A;
package B is
  type Something is record
    Field : A.Something;
  end record;
end B;

with B; -- no "with" of A
procedure Foo is
  X : B.Something;
begin
  X.Field := ...
```

Summary

Summary

- Library Units are "standalone" entities
  - Can contain subunits with similar structure
- `with` clauses interconnect library units
  - Express dependencies of the one being compiled
  - Not textual inclusion!

Packages

Introduction

Packages

- Enforce separation of client from implementation
  - In terms of compile-time visibility
  - For data
  - For type representation, when combined with `private` types
    - Abstract Data Types
- Provide basic namespace control
- Directly support software engineering principles
  - Especially in combination with `private` types
  - Modularity
  - Information Hiding (Encapsulation)
  - Abstraction
  - Separation of Concerns

Separating Interface and Implementation

- *Implementation* and *specification* are textually distinct from each other
  - Typically in separate files
- Clients can compile their code before body exists
  - All they need is the package specification
  - Clients have **no** visibility over the body
  - Full client/interface consistency is guaranteed

```
package Float_Stack is
  Max : constant := 100;
  procedure Push (X : in Float);
  procedure Pop (X : out Float);
end Float_Stack;
```

Uncontrolled Visibility Problem

- Clients have too much access to representation
  - Data
  - Type representation
- Changes force clients to recode and retest
- Manual enforcement is not sufficient
- Why fixing bugs introduces new bugs!

Basic Syntax and Nomenclature

```
package_declaration ::= package_specification;
```

- Spec

```
package_specification ::=
   package name is
      {basic_declarative_item}
   end [name];
```

- Body

```
package_body ::=
   package body name is
      declarative_part
   end [name];
```

Declarations

Package Declarations

- Required in all cases
  - Cannot have a package without the declaration
- Describe the client's interface
  - Declarations are exported to clients
  - Effectively the "pin-outs" for the black-box
- When changed, requires clients recompilation
  - The "pin-outs" have changed

```
package Float_Stack is
  Max : constant := 100;
  procedure Push (X : in Float);
  procedure Pop (X : out Float);
end Float_Stack;

package Data is
   Object : Integer;
end Data;
```

Compile-Time Visibility Control

- Items in the declaration are visible to users

```
package name  is
   -- exported declarations of
   --    types, variables, subprograms ...
end name;
```

- Items in the body are never externally visible
  - Compiler prevents external references

```
package body name  is
   -- hidden declarations of
   --    types, variables, subprograms ...
   -- implementations of exported subprograms etc.
end name;
```

Example of Exporting To Clients

- Variables, types, exception, subprograms, etc.
    - The primary reason for separate subprogram declarations

```
package P is
   procedure This_Is_Exported;
end P;

package body P is
   procedure Not_Exported is
      ...
   procedure This_Is_Exported is
      ...
end P;
```

Referencing Exported Items

- Achieved via "dot notation"
- Package Specification

```
package Float_Stack is
  Max : constant := 100;
  procedure Push (X : in Float);
  procedure Pop (X : out Float);
end Float_Stack;
```

- Package Reference

```
with Float_Stack;
procedure Test is
   X : Float;
begin
   Float_Stack.Pop (X);
   Float_Stack.Push (12.0);
   if Count < Float_Stack.Max then ...
```

Bodies

Package Bodies

- Dependent on corresponding package specification
  - Obsolete if specification changed
- Clients need only to relink if body changed
  - Any code that would require editing would not have compiled in the first place
- Necessary for specifications that require a completion, for example:
  - Subprogram bodies
  - Task bodies
  - Incomplete types in `private` part
  - Others...

Bodies Are Never Optional

- Either required for a given spec or not allowed at all
  - Based on declarations in that spec
- A change from Ada 83
- A (nasty) justification example will be shown later

Example Spec That Cannot Have A Body

```
package Graphics_Primitives is
  type Coordinate is digits 12;
  type Device_Coordinates is record
    X, Y : Integer;
  end record;
  type Normalized_Coordinates is record
    X, Y : Coordinate range 0.0 .. 1.0;
  end record;
  type Offset is record
    X, Y : Coordinate range -1.0 .. 1.0;
  end record;
  -- nothing to implement, so no body allowed
end Graphics_Primitives;
```

Example Spec Requiring A Package Body

```
package VT100 is
  subtype Rows is Integer range 1 .. 24;
  subtype Columns is Integer range 1 .. 80;
  type Position is record
    Row   : Rows := Rows'First;
    Col : Columns := Columns'First;
  end record;
   -- The following need to be defined in the body
  procedure Move_Cursor (To : in Position);
  procedure Home;
  procedure Clear_Screen;
  procedure Cursor_Up (Count : in Positive := 1);
end VT100;
```

Required Body Example

```
package body VT100 is
  -- This function is not visible outside this package
  function Unsigned (Input : Integer) return String is
    Str : constant String := Integer'Image (Input);
  begin
    return Str (2 .. Str'length);
  end Unsigned;
  procedure Move_Cursor (To : in Position) is
  begin
    Text_IO.Put (ASCII.Esc & 'I' &
                 Unsigned(To.Row) & ';' &
                 Unsigned(To.Col) & 'H');
  end Move_Cursor;
  procedure Home is
  begin
    Text_IO.Put (ASCII.Esc & "iH");
  end Home;
  procedure Cursor_Up (Count : in Positive := 1) is ...
    ...
end VT100;
```

Quiz

```
package P is
   Object_One : Integer;
   procedure One (P : out Integer);
end P;
```

Which completion(s) is(are) correct for `package P`?

A. `No completion is needed`

B.
```
package body P is
  procedure One (P : out Integer) is null;
end P;
```

C.
```
package body P is
  Object_One : Integer;
  procedure One (P : out Integer) is
  begin
    P := Object_One;
  end One;
end P;
```

D.
```
package body P is
  procedure One (P : out Integer) is
  begin
    P := Object_One;
  end One;
end P;
```

Quiz

```
package P is
   Object_One : Integer;
   procedure One (P : out Integer);
end P;
```

Which completion(s) is(are) correct for `package P`?

A. `No completion is needed`

B. *package body P is*
```
  procedure One (P : out Integer) is null;
end P;
```

C.
```
package body P is
  Object_One : Integer;
  procedure One (P : out Integer) is
  begin
    P := Object_One;
  end One;
end P;
```

D. *package body P is*
```
  procedure One (P : out Integer) is
  begin
    P := Object_One;
  end One;
end P;
```

A. Procedure One must have a body
B. Parameter P is `out` but not assigned (legal but not a good idea)
C. Redeclaration of Object_One
D. Correct

Executable Parts

Optional Executable Part

```
package_body ::=
    package body name is
       declarative_part
    [ begin
       handled_sequence_of_statements ]
    end [ name ];
```

Executable Part Semantics

- Executed only once, when package is elaborated
- Ideal when statements are required for initialization
  - Otherwise initial values in variable declarations would suffice

```
package body Random is
  Seed1, Seed2 : Integer;
  Call_Count : Natural := 0;
  procedure Initialize (Seed1 : out Integer;
                        Seed2 : out Integer) is ...
  function Number return Float is ...
begin -- Random
  Initialize (Seed1, Seed2);
end Random;
```

Requiring/Rejecting Bodies Justification

- Consider the alternative: an optional package body that becomes obsolete prior to building
- Builder could silently choose not to include the package in executable
  - Package executable part might do critical initialization!

```
package P is
  Data : array (L .. U) of
      Integer;
end P;

package body P is
  ...
begin
  for K in Data'Range loop
    Data(K) := ...
  end loop;
end P;
```

Forcing A Package Body To be Required

- Use `pragma Elaborate_Body`
  - Says to elaborate body immediately after spec
  - Hence there must be a body!
- Additional pragmas we will examine later

```
package P is
  pragma Elaborate_Body;
  Data : array (L .. U) of
      Integer;
end P;

package body P is
  ...
begin
  for K in Data'Range loop
    Data(K) := ...
  end loop;
end P;
```

Idioms

Named Collection of Declarations

- Exports:
  - Objects (constants and variables)
  - Types
  - Exceptions
- Does not export operations

```
package Physical_Constants is
  Polar_Radius_in_feet    : constant := 20_856_010.51;
  Equatorial_Radius_in_feet : constant := 20_926_469.20;
  Earth_Diameter_in_feet : constant := 2.0 *
     ((Polar_Radius_in_feet + Equatorial_Radius_in_feet)/2.0);
  Sea_Level_Air_Density : constant := 0.00239; --slugs/foot**3
  Altitude_Of_Tropopause_in_feet : constant := 36089.0;
  Tropopause_Temperature_in_celsius : constant := -56.5;
end Physical_Constants;
```

Named Collection of Declarations (2)

- Effectively application global data

```
package Equations_of_Motion is
  Longitudinal_Velocity : Float := 0.0;
  Longitudinal_Acceleration : Float := 0.0;
  Lateral_Velocity  : Float := 0.0;
  Lateral_Acceleration : Float := 0.0;
  Vertical_Velocity : Float := 0.0;
  Vertical_Acceleration : Float := 0.0;
  Pitch_Attitude : Float := 0.0;
  Pitch_Rate : Float := 0.0;
  Pitch_Acceleration : Float := 0.0;
end Equations_of_Motion;
```

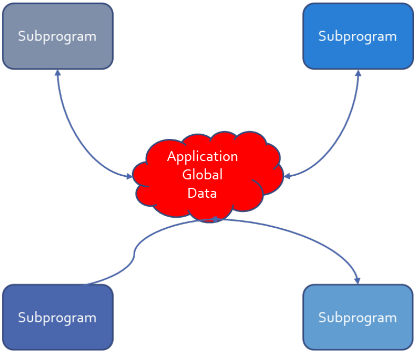
Group of Related Program Units

- Exports:
  - Objects
  - Types
  - Values
  - Operations
- Users have full access to type representations
  - This visibility may be necessary

```
package Linear_Algebra is
  type Vector is array (Positive range <>) of Float;
  function "+" (L,R : Vector) return Vector;
  function "*" (L,R : Vector) return Vector;
  ...
end Linear_Algebra;
```

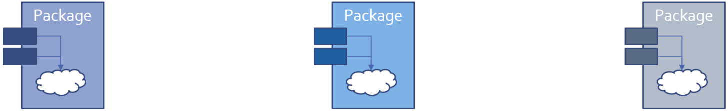
Uncontrolled Data Visibility Problem

- Effects of changes are potentially pervasive so one must understand everything before changing anything



Controlling Data Visibility Using Packages

- Divides global data into separate package bodies
- Visible only to procedures and functions declared in those same packages
  - Clients can only call these visible routines
- Global change effects are much less likely
  - Direct breakage is impossible



Abstract Data Machines

- Exports:
  - Operations
  - State information queries (optional)
- No direct user access to data

```
package Float_Stack is
  Max : constant := 100;
  procedure Push (X : in Float);
  procedure Pop (X : out Float);
end Float_Stack;

package body Float_Stack is
  type Contents is array (1 .. Max) of Float;
  Values : Contents;
  Top : Integer range 0 .. Max := 0;
  procedure Push (X : in Float) is ...
  procedure Pop (X : out Float) is ...
end Float_Stack;
```

Controlling Type Representation Visibility

- In other words, support for Abstract Data Types
  - No operations visible to clients based on representation
- The fundamental concept for Ada
- Requires `private` types discussed in coming section...

Lab

Packages Lab

- Requirements
  - Create a program to add and remove integer values from a list
  - Program should allow user to do the following as many times as desired
    - Add an integer in a pre-defined range to the list
    - Remove all occurrences of an integer from the list
    - Print the values in the list
- Hints
  - Create (at least) three packages
    1. minimum/maximum integer values and maximum number of items in list
    2. User input (ensure value is in range)
    3. List Abstract Data Machine
  - Remember: `with package_name;` gives access to `package_name`

Creating Packages in GNAT Studio

- Right-click on the source directory node
  - If you used a prompt, the directory is probably **.**
  - If you used the wizard, the directory is probably **src**
- New → Ada Package
  - Fill in name of Ada package
  - Check the box if you want to create the package body in addition to the package spec

Packages Lab Solution - Constants

```
package Constants is

   Lowest_Value  : constant := 100;
   Highest_Value : constant := 999;
   Maximum_Count : constant := 10;
   subtype Integer_T is Integer
      range Lowest_Value .. Highest_Value;

end Constants;
```

Packages Lab Solution - Input

```
with Constants;
package Input is
   function Get_Value (Prompt : String) return Constants.Integer_T;
end Input;

with Ada.Text_IO; use Ada.Text_IO;
package body Input is

   function Get_Value (Prompt : String) return Constants.Integer_T is
      Ret_Val : Integer;
   begin
      Put (Prompt & "> ");
      loop
         Ret_Val := Integer'Value (Get_Line);
         exit when Ret_Val >= Constants.Lowest_Value
           and then Ret_Val <= Constants.Highest_Value;
         Put ("Invalid. Try Again >");
      end loop;
      return Ret_Val;
   end Get_Value;

end Input;
```

Packages Lab Solution - List

```ada
package List is
   procedure Add (Value : Integer);
   procedure Remove (Value : Integer);
   function Length return Natural;
   procedure Print;
end List;

with Ada.Text_IO; use Ada.Text_IO;
with Constants;
package body List is
   Content : array (1 .. Constants.Maximum_Count) of Integer;
   Last    : Natural := 0;

   procedure Add (Value : Integer) is
   begin
      if Last < Content'Last then
         Last           := Last + 1;
         Content (Last) := Value;
      else
         Put_Line ("Full");
      end if;
   end Add;

   procedure Remove (Value : Integer) is
      I : Natural := 1;
   begin
      while I <= Last loop
         if Content (I) = Value then
            Content (I .. Last - 1) := Content (I + 1 .. Last);
            Last                    := Last - 1;
         else
            I := I + 1;
         end if;
      end loop;
   end Remove;

   procedure Print is
   begin
      for I in 1 .. Last loop
         Put_Line (Integer'Image (Content (I)));
      end loop;
   end Print;

   function Length return Natural is (Last);
end List;
```

Packages Lab Solution - Main

```
with Ada.Text_IO; use Ada.Text_IO;
with Input;
with List;
procedure Main is

begin

   loop
      Put ("(A)dd | (R)emove | (P)rint | Q(uit) : ");
      declare
         Str : constant String := Get_Line;
      begin
         exit when Str'Length = 0;
         case Str (Str'First) is
            when 'A' =>
               List.Add (Input.Get_Value ("Value to add"));
            when 'R' =>
               List.Remove (Input.Get_Value ("Value to remove"));
            when 'P' =>
               List.Print;
            when 'Q' =>
               exit;
            when others =>
               Put_Line ("Illegal entry");
         end case;
      end;
   end loop;

end Main;
```

Summary

Summary

- Emphasizes separations of concerns
- Solves the global visibility problem
  - Only those items in the specification are exported
- Enforces software engineering principles
  - Information hiding
  - Abstraction
- Implementation can't be corrupted by clients
  - Compiler won't let clients compile references to internals
- Bugs must be in the implementation, not clients
  - Only body implementation code has to be understood

Private Types

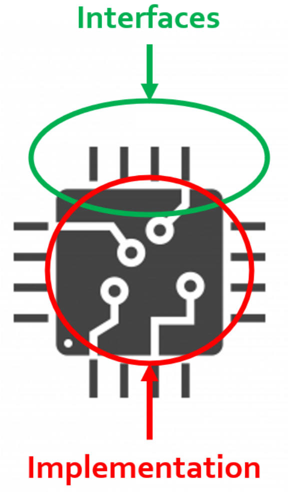
Introduction

Introduction

- Why does fixing bugs introduce new ones?
- Control over visibility is a primary factor
  - Changes to an abstraction's internals shouldn't break users
  - Including type representation
- Need tool-enforced rules to isolate dependencies
  - Between implementations of abstractions and their users
  - In other words, "information hiding"

Information Hiding

- A design technique in which implementation artifacts are made inaccessible to users
- Based on control of visibility to those artifacts
  - A product of "encapsulation"
  - Language support provides rigor
- Concept is "software integrated circuits"



Views

- Specify legal manipulation for objects of a type
  - Types are characterized by permitted values and operations
- Some views are implicit in language
  - Mode `in` parameters have a view disallowing assignment
- Views may be explicitly specified
  - Disallowing access to representation
  - Disallowing assignment
- Purpose: control usage in accordance with design
  - Adherence to interface
  - Abstract Data Types

Implementing Abstract Data Types via Views

Implementing Abstract Data Types

- A combination of constructs in Ada
- Not based on single "class" construct, for example
- Constituent parts
  - Packages, with "private part" of package spec
  - "Private types" declared in packages
  - Subprograms declared within those packages

Package Visible and Private Parts for Views

- Declarations in visible part are exported to users
- Declarations in private part are hidden from users
  - No compilable references to type's actual representation

```
package name is
... exported declarations of types, variables, subprograms ...
private
... hidden declarations of types, variables, subprograms ...
end name;
```

Declaring Private Types for Views

- Partial syntax

```
type defining_identifier is private;
```

- Private type declaration must occur in visible part
  - *Partial view*
  - Only partial information on the type
  - Users can reference the type name
    - But cannot create an object of that type until after the full type declaration
- Full type declaration must appear in private part
  - Completion is the *Full view*
  - **Never** visible to users
  - **Not** visible to designer until reached

```
package Control is
  type Valve is private;
  procedure Open (V : in out Valve);
  procedure Close (V : in out Valve);
  ...
private
  type Valve is ...
end Control;
```

Partial and Full Views of Types

- Private type declaration defines a *partial view*
  - The type name is visible
  - Only designer's operations and some predefined operations
  - No references to full type representation
- Full type declaration defines the *full view*
  - Fully defined as a record type, scalar, imported type, etc...
  - Just an ordinary type within the package
- Operations available depend upon one's view

Software Engineering Principles

- Encapsulation and abstraction enforced by views
    - Compiler enforces view effects
- Same protection as hiding in a package body
    - Recall "Abstract Data Machines" idiom
- Additional flexibility of types
    - Unlimited number of objects possible
    - Passed as parameters
    - Components of array and record types
    - Dynamically allocated
    - et cetera

Users Declare Objects of the Type

- Unlike "abstract data machine" approach
- Hence must specify which stack to manipulate
  - Via parameter

```
X, Y, Z : Stack;
...
Push (42, X);
...
if Empty (Y) then
...
Pop (Counter, Z);
```

Compile-Time Visibility Protection

- No type representation details available outside the package
- Therefore users cannot compile code referencing representation
- This does not compile

```
with Bounded_Stacks;
procedure User is
  S : Bounded_Stacks.Stack;
begin
  S.Top := 1;  -- Top is not visible
end User;
```

Benefits of Views

- Users depend only on visible part of specification
  - Impossible for users to compile references to private part
  - Physically seeing private part in source code is irrelevant
- Changes to implementation don't affect users
  - No editing changes necessary for user code
- Implementers can create bullet-proof abstractions
  - If a facility isn't working, you know where to look
- Fixing bugs is less likely to introduce new ones

Quiz

```
package P is
   type Private_T is private;

   type Record_T is record
```

Which component is legal?

A. `Field_A : Integer := Private_T'Pos (Private_T'First);`

B. `Field_B : Private_T := null;`

C. `Field_C : Private_T := 0;`

D. `Field_D : Integer := Private_T'Size;`

```
   end record;
```

Quiz

```
package P is
   type Private_T is private;

   type Record_T is record
```

Which component is legal?

A. `Field_A : Integer := Private_T'Pos (Private_T'First);`

B. `Field_B : Private_T := null;`

C. `Field_C : Private_T := 0;`

D. *`Field_D : Integer := Private_T'Size;`*

```
    end record;
```

Explanations

A. Visible part does not know `Private_T` is discrete
B. Visible part does not know possible values for `Private_T`
C. Visible part does not know possible values for `Private_T`
D. Correct - type will have a known size at run-time

Private Part Construction

Private Part Location

- Must be in package specification, not body
- Body usually compiled separately after declaration
- Users can compile their code before the package body is compiled or even written
    - Package definition

```
package Bounded_Stacks is
  type Stack is private;
  ...
private
  type Stack is ...
end Bounded_Stacks;
```

    - Package reference

```
with Bounded_Stacks;
procedure User is
  S : Bounded_Stacks.Stack;
...
begin
  ...
end User;
```

Private Part and Recompilation

- Private part is part of the specification
  - Compiler needs info from private part for users' code, e.g., storage layouts for private-typed objects
- Thus changes to private part require user recompilation
- Some vendors avoid "unnecessary" recompilation
  - Comment additions or changes
  - Additions which nobody yet references

Declarative Regions

- Declarative region of the spec extends to the body
  - Anything declared there is visible from that point down
  - Thus anything declared in specification is visible in body

```
package Foo is
   type Private_T is private;
   procedure X (B : in out Private_T);
private
   -- Y and Hidden_T are not visible to users
   procedure Y (B : in out Private_T);
   type Hidden_T is ...;
   type Private_T is array (1 .. 3) of Hidden_T;
end Foo;

package body Foo is
   -- Z is not visible to users
   procedure Z (B : in out Private_T) is ...
   procedure Y (B : in out Private_T) is ...
   procedure X (B : in out Private_T) is ...
 end Foo;
```

Full Type Declaration

- May be any type
  - Predefined or user-defined
  - Including references to imported types
- Contents of private part are unrestricted
  - Anything a package specification may contain
  - Types, subprograms, variables, etc.

```
package P is
  type T is private;
  ...
private
  type Vector is array (1.. 10)
     of Integer;
  function Initial
     return Vector;
  type T is record
    A, B : Vector := Initial;
  end record;
end P;
```

Deferred Constants

- Visible constants of a hidden representation
    - Value is "deferred" to private part
    - Value must be provided in private part
- Not just for private types, but usually so

```
package P is
  type Set is private;
  Null_Set : constant Set; -- exported name
  ...
private
  type Index is range ...
  type Set is array (Index) of Boolean;
  Null_Set : constant Set :=  -- definition
     (others => False);
end P;
```

Quiz

```
package P is
   type Private_T is private;
   Object_A : Private_T;
   procedure Proc (Param : in out Private_T);
private
   type Private_T is new Integer;
   Object_B : Private_T;
end package P;

package body P is
   Object_C : Private_T;
   procedure Proc (Param : in out Private_T) is null;
end P;
```

Which object definition is **not** legal?

A. `Object_A`
B. `Object_B`
C. `Object_C`
D. None of the above

Quiz

```
package P is
   type Private_T is private;
   Object_A : Private_T;
   procedure Proc (Param : in out Private_T);
private
   type Private_T is new Integer;
   Object_B : Private_T;
end package P;

package body P is
   Object_C : Private_T;
   procedure Proc (Param : in out Private_T) is null;
end P;
```

Which object definition is **not** legal?

A. *`Object_A`*
B. `Object_B`
C. `Object_C`
D. None of the above

An object cannot be declared until its type is fully declared. `Object_A` could be declared constant, but then it would have to be finalized in the `private` section.

View Operations

View Operations

- A matter of inside versus outside the package
  - Inside the package the view is that of the designer
  - Outside the package the view is that of the user

- **User** of package has **Partial** view
  - Operations exported by package
  - Basic operations

- **Designer** of package has **Full** view
  - **Once** completion is reached
  - All operations based upon full definition of type
  - Indexed components for arrays
  - components for records
  - Type-specific attributes
  - Numeric manipulation for numerics
  - et cetera

Designer View Sees Full Declaration

```
package Bounded_Stacks is
  Capacity : constant := 100;
  type Stack is private;
  procedure Push (Item : in Integer; Onto : in out Stack);
  ...
private
  type Index is range 0 .. Capacity;
  type Vector is array (Index range 1..Capacity) of Integer;
  type Stack is record
     Top : Integer;
     ...
end Bounded_Stacks;
```

Designer View Allows All Operations

```
package body Bounded_Stacks is
  procedure Push (Item : in Integer;
                  Onto : in out Stack) is
  begin
     Onto.Top := Onto.Top + 1;
     ...
  end Push;

  procedure Pop (Item : out Integer;
                 From : in out Stack) is
  begin
     Onto.Top := Onto.Top - 1;
     ...
  end Pop;
end Bounded_Stacks;
```

Users Have the Partial View

- Since they are outside package
- Basic operations
- Exported subprograms

```
package Bounded_Stacks is
  type Stack is private;
  procedure Push (Item : in Integer; Onto : in out Stack);
  procedure Pop (Item : out Integer; From : in out Stack);
  function Empty (S : Stack) return Boolean;
  procedure Clear (S : in out Stack);
  function Top (S : Stack) return Integer;
private
  ...
end Bounded_Stacks;
```

User View's Activities

- Declarations of objects
  - Constants and variables
  - Must call designer's functions for values

```
C : Complex.Number := Complex.I;
```

- Assignment, equality and inequality, conversions
- Designer's declared subprograms
- User-declared subprograms
  - Using parameters of the exported private type
  - Dependent on designer's operations

User View Formal Parameters

- Dependent on designer's operations for manipulation
  - Cannot reference type's representation
- Can have default expressions of private types

```
-- external implementation of "Top"
procedure Get_Top (
    The_Stack : in out Bounded_Stacks.Stack;
    Value : out Integer) is
  Local : Integer;
begin
  Bounded_Stacks.Pop (Local, The_Stack);
  Value := Local;
  Bounded_Stacks.Push (Local, The_Stack);
end Get_Top;
```

Limited Private

- `limited` is itself a view
  - Cannot perform assignment, copy, or equality
- `limited private` can restrain user's operation
  - Actual type **does not** need to be `limited`

```
package UART is
    type Instance is limited private;
    function Get_Next_Available return Instance;
[...]

declare
   A, B := UART.Get_Next_Available;
begin
   if A = B -- Illegal
   then
       A := B; -- Illegal
   end if;
```

When To Use or Avoid Private Types

When To Use Private Types

- Implementation may change
  - Allows users to be unaffected by changes in representation
- Normally available operations do not "make sense"
  - Normally available based upon type's representation
  - Determined by intent of ADT

```
A : Valve;
B : Valve;
C : Valve;
...
C := A + B;   -- addition not meaningful
```

- Users have no "need to know"
  - Based upon expected usage

When To Avoid Private Types

- If the abstraction is too simple to justify the effort
  - But that's the thinking that led to Y2K rework
- If normal user interface requires representation-specific operations that cannot be provided
  - Those that cannot be redefined by programmers
  - Would otherwise be hidden by a private type
  - If **Vector** is private, indexing of elements is annoying

```
type Vector is array (Positive range <>) of Float;
V : Vector (1 .. 3);
...
V (1) := Alpha;
```

Idioms

Effects of Hiding Type Representation

- Makes users independent of representation
  - Changes cannot require users to alter their code
  - Software engineering is all about money...
- Makes users dependent upon exported operations
  - Because operations requiring representation info are not available to users
    - Expression of values (aggregates, etc.)
    - Assignment for limited types
- Common idioms are a result
  - *Constructor*
  - *Selector*

Constructors

- Create designer's objects from user's values
- Usually functions

```
package Complex is
  type Number is private;
  function Make (Real_Part : Float; Imaginary : Float) return Number
private
  type Number is record ...
end Complex;

package body Complex is
   function Make (Real_Part : Float; Imaginary_Part : Float)
     return Number is ...
end Complex:
...
A : Complex.Number :=
    Complex.Make (Real_Part => 2.5, Imaginary => 1.0);
```

Procedures As Constructors

- Spec

```
package Complex is
  type Number is private;
  procedure Make (This : out Number;  Real_Part, Imaginary : in Float) ;
  ...
private
  type Number is record
    Real_Part, Imaginary : Float;
  end record;
end Complex;
```

- Body (partial)

```
package body Complex is
  procedure Make (This : out Number;
                  Real_Part, Imaginary : in Float) is
    begin
      This.Real_Part := Real_Part;
      This.Imaginary := Imaginary;
    end Make;
...
```

Selectors

- Decompose designer's objects into user's values
- Usually functions

```
package Complex is
  type Number is private;
  function Real_Part (This: Number) return Float;
  ...
private
  type Number is record
    Real_Part, Imaginary : Float;
  end record;
end Complex;

package body Complex is
  function Real_Part (This : Number) return Float is
  begin
    return This.Real_Part;
  end Real_Part;
  ...
end Complex;
...
Phase : Complex.Number := Complex.Make (10.0, 5.5);
Object : Float := Complex.Real_Part (Phase);
```

Lab

Private Types Lab

- Requirements
  - Implement a program to create a map such that
    - Map key is a description of a flag
    - Map element content is the set of colors in the flag
  - Operations on the map should include: Add, Remove, Modify, Get, Exists, Image
  - Main program should print out the entire map before exiting
- Hints
  - Should implement a **map** ADT (to keep track of the flags)
    - This **map** will contain all the flags and their color descriptions
  - Should implement a **set** ADT (to keep track of the colors)
    - This **set** will be the description of the map element
  - Each ADT should be its own package
  - At a minimum, the **map** and **set** type should be **private**

Private Types Lab Solution - Color Set

```
package Colors is
   type Color_T is (Red, Yellow, Green, Blue, Black);
   type Color_Set_T is private;

   Empty_Set : constant Color_Set_T;

   procedure Add (Set   : in out Color_Set_T;
                  Color :        Color_T);
   procedure Remove (Set   : in out Color_Set_T;
                     Color :        Color_T);
   function Image (Set : Color_Set_T) return String;
private
   type Color_Set_Array_T is array (Color_T) of Boolean;
   type Color_Set_T is record
      Values : Color_Set_Array_T := (others => False);
   end record;
   Empty_Set : constant Color_Set_T := (Values => (others => False));
end Colors;

package body Colors is
   procedure Add (Set   : in out Color_Set_T;
                  Color :        Color_T) is
   begin
      Set.Values (Color) := True;
   end Add;
   procedure Remove (Set   : in out Color_Set_T;
                     Color :        Color_T) is
   begin
      Set.Values (Color) := False;
   end Remove;

   function Image (Set   : Color_Set_T;
                   First : Color_T;
                   Last  : Color_T)
                   return String is
      Str : constant String := (if Set.Values (First) then Color_T'Image (First) else "");
   begin
      if First = Last then
         return Str;
      else
         return Str & " " & Image (Set, Color_T'Succ (First), Last);
      end if;
   end Image;
   function Image (Set : Color_Set_T) return String is
      (Image (Set, Color_T'First, Color_T'Last));
end Colors;
```

Private Types Lab Solution - Flag Map (Spec)

```
with Colors;
package Flags is
   type Key_T is (USA, England, France, Italy);
   type Map_Element_T is private;
   type Map_T is private;

   procedure Add (Map         : in out Map_T;
                  Key         :        Key_T;
                  Description :        Colors.Color_Set_T;
                  Success     :    out Boolean);
   procedure Remove (Map     : in out Map_T;
                     Key     :        Key_T;
                     Success :    out Boolean);
   procedure Modify (Map         : in out Map_T;
                     Key         :        Key_T;
                     Description :        Colors.Color_Set_T;
                     Success     :    out Boolean);

   function Exists (Map : Map_T; Key : Key_T) return Boolean;
   function Get (Map : Map_T; Key : Key_T) return Map_Element_T;
   function Image (Item : Map_Element_T) return String;
   function Image (Flag : Map_T) return String;
private
   type Map_Element_T is record
      Key         : Key_T := Key_T'First;
      Description : Colors.Color_Set_T := Colors.Empty_Set;
   end record;
   type Map_Array_T is array (1 .. 100) of Map_Element_T;
   type Map_T is record
      Values : Map_Array_T;
      Length : Natural := 0;
   end record;
end Flags;
```

Private Types Lab Solution - Flag Map (Body - 1 of 2)

```
procedure Add (Map         : in out Map_T;
               Key         :        Key_T;
               Description :        Colors.Color_Set_T;
               Success     :    out Boolean) is
begin
   Success := (for all Item of Map.Values
        (1 .. Map.Length) => Item.Key /= Key);
   if Success then
      declare
         New_Item : constant Map_Element_T :=
           (Key => Key, Description => Description);
      begin
         Map.Length              := Map.Length + 1;
         Map.Values (Map.Length) := New_Item;
      end;
   end if;
end Add;
procedure Remove (Map     : in out Map_T;
                  Key     :        Key_T;
                  Success :    out Boolean) is
begin
   Success := False;
   for I in 1 .. Map.Length loop
      if Map.Values (I).Key = Key then
         Map.Values
           (I .. Map.Length - 1) := Map.Values
             (I + 1 .. Map.Length);
         Map.Length := Map.Length - 1;
         Success := True;
         exit;
      end if;
   end loop;
end Remove;
```

Private Types Lab Solution - Flag Map (Body - 2 of 2)

```
   procedure Modify (Map         : in out Map_T;
                     Key         :        Key_T;
                     Description :        Colors.Color_Set_T;
                     Success     :    out Boolean) is
   begin
      Success := False;
      for I in 1 .. Map.Length loop
         if Map.Values (I).Key = Key then
            Map.Values (I).Description := Description;
            Success                    := True;
            exit;
         end if;
      end loop;
   end Modify;
   function Exists (Map : Map_T; Key : Key_T) return Boolean is
      (for some Item of Map.Values (1 .. Map.Length) => Item.Key = Key);
   function Get (Map : Map_T; Key : Key_T) return Map_Element_T is
      Ret_Val : Map_Element_T;
   begin
      for I in 1 .. Map.Length loop
         if Map.Values (I).Key = Key then
            Ret_Val := Map.Values (I);
            exit;
         end if;
      end loop;
      return Ret_Val;
   end Get;
   function Image (Item : Map_Element_T) return String is
     (Key_T'Image (Item.Key) & " => " & Colors.Image (Item.Description));
   function Image (Flag : Map_T) return String is
      Ret_Val : String (1 .. 1_000);
      Next    : Integer := Ret_Val'First;
   begin
      for Item of Flag.Values (1 .. Flag.Length) loop
         declare
            Str : constant String := Image (Item);
         begin
            Ret_Val (Next .. Next + Str'Length) := Image (Item) & ASCII.LF;
            Next                             := Next + Str'Length + 1;
         end;
      end loop;
      return Ret_Val (1 .. Next - 1);
   end Image;
```

Private Types Lab Solution - Main

```
with Ada.Text_IO; use Ada.Text_IO;
with Colors;
with Flags;
with Input;
procedure Main is
   Map : Flags.Map_T;
begin

   loop
      Put ("Enter country name (");
      for Key in Flags.Key_T loop
         Put (Flags.Key_T'Image (Key) & " ");
      end loop;
      Put ("): ");
      declare
         Str         : constant String := Get_Line;
         Key         : Flags.Key_T;
         Description : Colors.Color_Set_T;
         Success     : Boolean;
      begin
         exit when Str'Length = 0;
         Key         := Flags.Key_T'Value (Str);
         Description := Input.Get;
         if Flags.Exists (Map, Key) then
            Flags.Modify (Map, Key, Description, Success);
         else
            Flags.Add (Map, Key, Description, Success);
         end if;
      end;
   end loop;

   Put_Line (Flags.Image (Map));
end Main;
```

Summary

Summary

- Tool-enforced support for Abstract Data Types
  - Same protection as Abstract Data Machine idiom
  - Capabilities and flexibility of types
- May also be `limited`
  - Thus additionally no assignment or predefined equality
  - More on this later
- Common interface design idioms have arisen
  - Resulting from representation independence
- Assume private types as initial design choice
  - Change is inevitable

Program Structure

Introduction

Introduction

- Moving to "bigger" issues of overall program composition
- How to compose programs out of program units
- How to control object lifetimes
- How to define subsystems

Building A System

What is a System?

- Also called Application or Program or ...
- Collection of *library units*
  - Which are a collection of packages, subprograms, objects

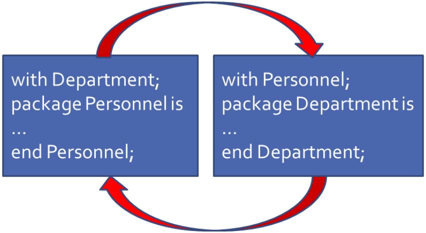
Library Units Review

- Those units not nested within another program unit
- Candidates
  - Subprograms
  - Packages
  - Generic Units
  - Generic Instantiations
  - Renamings
- Dependencies between library units via `with` clauses
  - What happens when two units need to depend on each other?

Circular Dependencies

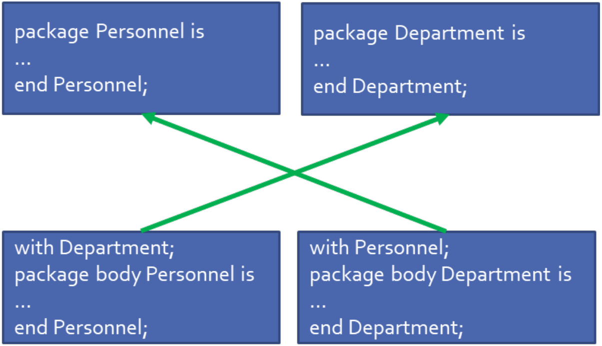
Handling Cyclic Dependencies

- Elaboration must be linear
- Package declarations cannot depend on each other
  - No linear order is possible
- Which package elaborates first?



Body-Level Cross Dependencies Are OK

- The bodies only depend on other packages' declarations
- The declarations are already elaborated by the time the bodies are elaborated



Resulting Design Problem

- Good design dictates that conceptually distinct types appear in distinct package declarations
  - Separation of concerns
  - High level of *cohesion*
- Not possible if they depend on each other
- One solution is to combine them in one package, even though conceptually distinct
  - Poor software engineering
  - May be only choice, depending on language version
    - Best choice would be to implement both parts in a new package

Illegal Package Declaration Dependency

```
with Department;
package Personnel is
  type Employee is private;
  procedure Assign (This : in Employee;
                    To : in out Department.Section);
private
  type Employee is record
    Assigned_To : Department.Section;
  end record;
end Personnel;

with Personnel;
package Department is
  type Section is private;
  procedure Choose_Manager (This : in out Section;
                            Who : in Personnel.Employee);
private
  type Section is record
    Manager : Personnel.Employee;
  end record;
end Department;
```

limited with Clauses

Ada 2005

- Solve the cyclic declaration dependency problem
  - Controlled cycles are now permitted
- Provide a *limited view* of the specified package
  - Only type names are visible (including in nested packages)
  - Types are viewed as *incomplete types*
- Normal view

```
package Personnel is
  type Employee is private;
  procedure Assign ...
private
  type Employee is ...
end Personnel;
```

- Implied limited view

```
package Personnel is
  type Employee;
end Personnel;
```

Using Incomplete Types

- Anywhere that the compiler doesn't yet need to know how they are really represented
  - Access types designating them
  - Access parameters designating them
  - Anonymous access components designating them
  - As formal parameters and function results
    - As long as compiler knows them at the point of the call
  - As generic formal type parameters
  - As introductions of private types
- If `tagged`, may also use **'Class**
- Thus typically involves some advanced features

Legal Package Declaration Dependency

Ada 2005

```
limited with Department;
package Personnel is
  type Employee is private;
  procedure Assign (This : in Employee;
                    To : in out Department.Section);
private
  type Employee is record
    Assigned_To : access Department.Section;
  end record;
end Personnel;

limited with Personnel;
package Department is
  type Section is private;
  procedure Choose_Manager (This : in out Section;
                            Who : in Personnel.Employee);
private
  type Section is record
    Manager : access Personnel.Employee;
  end record;
end Department;
```

Full **with** Clause On the Package Body

Ada 2005

- Even though declaration has a `limited with` clause
- Typically necessary since body does the work
  - Dereferencing, etc.
- Usual semantics from then on

```
limited with Personnel;
package Department is
...
end Department;

with Personnel; -- normal view in body
package body Department is
...
end Department;
```

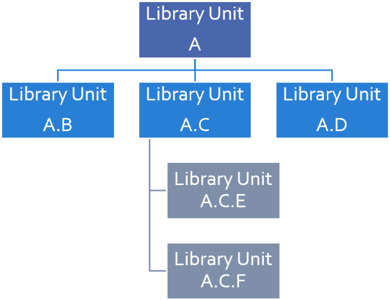
Hierarchical Library Units

Problem: Packages Are Not Enough

- Extensibility is a problem for private types
  - Provide excellent encapsulation and abstraction
  - But one has either complete visibility or essentially none
  - New functionality must be added to same package for sake of compile-time visibility to representation
  - Thus enhancements require editing/recompilation/retesting
- Should be something "bigger" than packages
  - Subsystems
  - Directly relating library items in one name-space
    - One big package has too many disadvantages
  - Avoiding name clashes among independently-developed code

Solution: Hierarchical Library Units

- Address extensibility issue
  - Can extend packages with visibility to parent private part
  - Extensions do not require recompilation of parent unit
  - Visibility of parent's private part is protected
- Directly support subsystems
  - Extensions all have the same ancestor *root* name



Programming By Extension

- *Parent unit*

```
package Complex is
  type Number is private;
  function "*" (Left, Right : Number) return Number;
  function "/" (Left, Right : Number) return Number;
  function "+" (Left, Right : Number) return Number;
  function "-" (Left, Right : Number) return Number;
...
private
  type Number is record
    Real_Part, Imaginary_Part : Float;
  end record;
end Complex;
```

- Extension created to work with parent unit

```
package Complex.Utils is
  procedure Put (C : in Number);
  function As_String (C : Number) return String;
  ...
end Complex.Utils;
```

Extension Can See Private Section

- With certain limitations

```
with Ada.Text_IO;
package body Complex.Utils is
  procedure Put(C : in Number) is
  begin
    Ada.Text_IO.Put(As_String(C));
  end Put;
  function As_String(C : Number) return String is
  begin
    -- Real_Part and Imaginary_Part are
    -- visible to child's body
    return "(" & Float'Image(C.Real_Part) & ", " &
           Float'Image(C.Imaginary_Part) & ")";
  end As_String;
...
end Complex.Utils;
```

Subsystem Approach

```
with Interfaces.C;
package OS is -- Unix and/or POSIX
 type File_Descriptor is new Interfaces.C.int;
  ...
end OS;

package OS.Mem_Mgmt is
  ...
  procedure Dump (File                 : File_Descriptor;
                  Requested_Location : System.Address;
                  Requested_Size     : Interfaces.C.Size_T);
  ...
end OS.Mem_Mgmt;

package OS.Files is
  ...
  function Open (Device : Interfaces.C.char_array;
                 Permission : Permissions := S_IRWXO)
                 return File_Descriptor;
  ...
end OS.Files;
```

Predefined Hierarchies

- Standard library facilities are children of **Ada**
    - **Ada.Text_IO**
    - **Ada.Calendar**
    - **Ada.Command_Line**
    - **Ada.Exceptions**
    - et cetera
- Other root packages are also predefined
    - **Interfaces.C**
    - **Interfaces.Fortran**
    - **System.Storage_Pools**
    - **System.Storage_Elements**
    - et cetera

Hierarchical Visibility

- Children can see ancestors' visible and private parts
  - All the way up to the root library unit
- Siblings have no automatic visibility to each other
- Visibility same as nested
  - As if child library units are nested within parents
    - All child units come after the root parent's specification
    - Grandchildren within children, great-grandchildren within ...

Example of Visibility As If Nested

```
package Complex is
  type Number is private;
  function "*" (Left, Right : Number) return Number;
  function "/" (Left, Right : Number) return Number;
  function "+" (Left, Right : Number) return Number;
  ...
private
  type Number is record
    Real_Part : Float;
    Imaginary : Float;
  end record;
  package Utils is
    procedure Put (C : in Number);
    function As_String (C : Number) return String;
    ...
  end Utils;
end Complex;
```

with Clauses for Ancestors are Implicit

- Because children can reference ancestors' private parts
  - Code is not in executable unless somewhere in the **with** clauses
- Explicit clauses for ancestors are redundant but OK

```
package Parent is
  ...
private
  A : Integer := 10;
end Parent;

-- no "with" of parent needed
package Parent.Child is
   ...
private
  B : Integer := Parent.A;
  -- no dot-notation needed
  C : Integer := A;
end Parent.Child;
```

with Clauses for Siblings are Required

- If references are intended

```
with A.Foo; --required
package body A.Bar is
   ...
   -- 'Foo' is directly visible because of the
   -- implied nesting rule
   X : Foo.Typemark;
end A.Bar;
```

Quiz

```
package Parent is
   Parent_Object : Integer;
end Parent;

package Parent.Sibling is
   Sibling_Object : Integer;
end Parent.Sibling;

package Parent.Child is
   Child_Object : Integer := ? ;
end Parent.Child;
```

Which is not a legal initialization of Child_Object?

A. `Parent.Parent_Object + Parent.Sibling.Sibling_Object`
B. `Parent_Object + Sibling.Sibling_Object`
C. `Parent_Object + Sibling_Object`
D. All of the above

Quiz

```
package Parent is
   Parent_Object : Integer;
end Parent;

package Parent.Sibling is
   Sibling_Object : Integer;
end Parent.Sibling;

package Parent.Child is
   Child_Object : Integer := ? ;
end Parent.Child;
```

Which is not a legal initialization of Child_Object?

A. `Parent.Parent_Object + Parent.Sibling.Sibling_Object`
B. `Parent_Object + Sibling.Sibling_Object`
C. `Parent_Object + Sibling_Object`
D. ***All of the above***

A, B, and C are illegal because there is no reference to package `Parent.Sibling` (the reference to `Parent` is implied by the hierarchy). If `Parent.Child` had "`with Parent.Sibling;`", then A and B would be legal, but C would still be incorrect because there is no implied reference to a sibling.

Visibility Limits

Parents Do Not Know Their Children!

- Children grant themselves access to ancestors' private parts
  - May be created well after parent
  - Parent doesn't know if/when child packages will exist
- Alternatively, language *could have* been designed to grant access when declared
  - Like `friend` units in C++
  - But would have to be prescient!
    - Or else adding children requires modifying parent
  - Hence too restrictive
- Note: Parent body can reference children
  - Typical method of parsing out complex processes

Correlation to C++ Class Visibility Controls

- Ada private part is visible to child units

```
package P is
  A ...
private
  B ...
end P;
package body P is
  C ...
end P;
```

- Thus private part is like the protected part in C++

```
class C {
public:
  A ...
protected:
  B ...
private:
  C ...
};
```

Visibility Limits

- Visibility to parent's private part is not open-ended
  - Only visible to private parts and bodies of children
  - As if only private part of child package is nested in parent
- Recall users can only reference exported declarations
  - Child public spec only has access to parent public spec

```
package Parent is
   ...
private
   type Parent_T is ...
end Parent;

package Parent.Child is
  -- Parent_T is not visible here!
private
  -- Parent_T is visible here
end Parent.Child;

package body Parent.Child is
 -- Parent_T is visible here
end Parent.Child;
```

Children Can Break Abstraction

- Could **break** a parent's abstraction
  - Alter a parent package state
  - Alters an ADT object state
- Useful for reset, testing: fault injections...

```
package Stack is
   ...
private
   Values : array (1 .. N) of Foo;
   Top : Natural range 0 .. N := 0;
end Stack;

package body Stack.Reset is
   procedure Reset is
   begin
     Top := 0;
   end Reset;
end Stack.Reset;
```

Using Children for Debug

- Provide **accessors** to parent's private information
- eg internal metrics...

```
package P is
   ...
private
  Internal_Counter : Integer := 0;
end P;
```

```
package P.Child is
  function Count return Integer;
end P.Child;
```

```
package body P.Child is
  function Count return Integer is
  begin
    return Internal_Counter;
  end Count;
end P.Child;
```

Quiz

```
package P is
   procedure Initialize;
   Object_A : Integer;
private
   Object_B : Integer;
end P;

package body P is
   Object_C : Integer;
   procedure Initialize is null;
end P;

package P.Child is
   function X return Integer;
end P.Child;
```

Which return statement would **not** be legal in P.Child.X?

A. `return Object_A;`
B. `return Object_B;`
C. `return Object_C;`
D. None of the above

Quiz

```
package P is
   procedure Initialize;
   Object_A : Integer;
private
   Object_B : Integer;
end P;

package body P is
   Object_C : Integer;
   procedure Initialize is null;
end P;

package P.Child is
   function X return Integer;
end P.Child;
```

Which return statement would **not** be legal in P.Child.X?

A. `return Object_A;`
B. `return Object_B;`
C. *`return Object_C;`*
D. None of the above

Explanations

A. `Object_A` is in the public part of `P` - visible to any unit that **`with`**'s `P`
B. `Object_B` is in the private part of `P` - visible in the private part or body of any descendant of `P`
C. `Object_C` is in the body of `P`, so it is only visible in the body of `P`
D. A and B are both valid completions

Private Children

Private Children

- Intended as implementation artifacts
- Only available within subsystem
    - Rules prevent **with** clauses by clients
    - Thus cannot export anything outside subsystem
    - Thus have no parent visibility restrictions
        - Public part of child also has visibility to ancestors' private parts

```
private package Maze.Debug is
   procedure Dump_State;
   ...
end Maze.Debug;
```

Rules Preventing Private Child Visibility

- Only available within immediate family
  - Rest of subsystem cannot import them
- Public unit declarations have import restrictions
  - To prevent re-exporting private information
- Public unit bodies have no import restrictions
  - Since can't re-export any imported info
- Private units can import anything
  - Declarations and bodies can import public and private units
  - Cannot be imported outside subsystem so no restrictions

Import Rules

- Only parent of private unit and its descendants can import a private child
- Public unit declarations import restrictions
  - Not allowed to have `with` clauses for private units
    - Exception explained in a moment
  - Precludes re-exporting private information
- Private units can import anything
  - Declarations and bodies can import private children

Some Public Children Are Trustworthy

- Would only use a private sibling's exports privately
- But rules disallow **with** clause

```
private package OS.UART is
 type Device is limited private;
 procedure Open (This : out Device; ...);
 ...
end OS.UART;

-- illegal - private child
with OS.UART;
package OS.Serial is
  type COM_Port is limited private;
  ...
private
  type COM_Port is limited record
    -- but I only need it here!
    COM : OS.UART.Device;
  ...
  end record;
end OS.Serial;
```

Solution 1: Move Type To Parent Package

```
package OS is
   ...
private
   -- no longer an ADT!
   type Device is limited private;
...
end OS;
private package OS.UART is
   procedure Open (This : out Device;
    ...);
   ...
end OS.UART;
```

```
package OS.Serial is
   type COM_Port is limited private;
   ...
private
   type COM_Port is limited record
     COM : Device; -- now visible
     ...
   end record;
end OS.Serial;
```

Solution 2: Partially Import Private Unit

Ada 2005

- Via **private with** clause
- Syntax

```
private with package_name {, package_name} ;
```

- Public declarations can then access private siblings
  - But only in their private part
  - Still prevents exporting contents of private unit
- The specified package need not be a private unit
  - But why bother otherwise

private with Example

Ada 2005

```
private package OS.UART is
  type Device is limited private;
  procedure Open (This : out Device;
     ...);
  ...
end OS.UART;
```

```
private with OS.UART;
package OS.Serial is
  type COM_Port is limited private;
  ...
private
  type COM_Port is limited record
    COM : OS.UART.Device;
    ...
  end record;
end OS.Serial;
```

Combining Private and Limited Withs

Ada 2005

- Cyclic declaration dependencies allowed
- A public unit can **with** a private unit
- With-ed unit only visible in the private part

```
limited with Parent.Public_Child;
private package Parent.Private_Child is
  type T is ...
end Parent.Private_Child;

limited private with Parent.Private_Child;
package Parent.Public_Child is
   ...
private
  X : access Parent.Private_Child.T;
end Parent.Public_Child;
```

Child Subprograms

- Child units can be subprograms
    - Recall syntax
    - Both public and private child subprograms
- Separate declaration required if private
    - Syntax doesn't allow `private` on subprogram bodies
- Only library packages can be parents
    - Only they have necessary scoping

```
private procedure Parent.Child;
```

Lab

Program Structure Lab

- Requirements
  - Create a message data type
    - Actual message type should be private
    - Need primitives to construct message and query contents
  - Create a child package that allows clients to modify the contents of the message
  - Main program should
    - Build a message
    - Print the contents of the message
    - Modify part of the message
    - Print the new contents of the message
- **Note: There is no prompt for this lab - you need to learn how to build the program structure**

Program Structure Lab Solution - Messages

```
package Messages is
   type Message_T is private;
   type Kind_T is (Command, Query);
   type Request_T is digits 6;
   type Status_T is mod 255;

   function Create (Kind    : Kind_T;
                    Request : Request_T;
                    Status  : Status_T)
                    return Message_T;

   function Kind (Message : Message_T) return Kind_T;
   function Request (Message : Message_T) return Request_T;
   function Status (Message : Message_T) return Status_T;

private
   type Message_T is record
      Kind    : Kind_T;
      Request : Request_T;
      Status  : Status_T;
   end record;
end Messages;

package body Messages is

   function Create (Kind    : Kind_T;
                    Request : Request_T;
                    Status  : Status_T)
                    return Message_T is
      (Kind => Kind, Request => Request, Status  => Status);

   function Kind (Message : Message_T) return Kind_T is
      (Message.Kind);
   function Request (Message : Message_T) return Request_T is
      (Message.Request);
   function Status (Message : Message_T) return Status_T is
      (Message.Status);

end Messages;
```

Program Structure Lab Solution - Message Modification

```
package Messages.Modify is

   procedure Kind (Message   : in out Message_T;
                   New_Value :        Kind_T);
   procedure Request (Message   : in out Message_T;
                      New_Value :        Request_T);
   procedure Status (Message   : in out Message_T;
                     New_Value :        Status_T);

end Messages.Modify;

package body Messages.Modify is

   procedure Kind (Message   : in out Message_T;
                   New_Value :        Kind_T) is
   begin
      Message.Kind := New_Value;
   end Kind;

   procedure Request (Message   : in out Message_T;
                      New_Value :        Request_T) is
   begin
      Message.Request := New_Value;
   end Request;

   procedure Status (Message   : in out Message_T;
                     New_Value :        Status_T) is
   begin
      Message.Status := New_Value;
   end Status;

end Messages.Modify;
```

Program Structure Lab Solution - Main

```
with Ada.Text_IO; use Ada.Text_IO;
with Messages;
with Messages.Modify;
procedure Main is
   Message : Messages.Message_T;
   procedure Print is
   begin
      Put_Line ("Kind => " & Messages.Kind (Message)'Image);
      Put_Line ("Request => " & Messages.Request (Message)'Image);
      Put_Line ("Status => " & Messages.Status (Message)'Image);
      New_Line;
   end Print;
begin
   Message := Messages.Create (Kind    => Messages.Command,
                               Request => 12.34,
                               Status  => 56);
   Print;
   Messages.Modify.Request (Message   => Message,
                            New_Value => 98.76);
   Print;
end Main;
```

Summary

Summary

- Hierarchical library units address important issues
  - Direct support for subsystems
  - Extension without recompilation
  - Separation of concerns with controlled sharing of visibility (Ada 2012)
- Parents should document assumptions for children
  - "These must always be in ascending order!"
- Children cannot misbehave unless imported ("with'ed")
- The writer of a child unit must be trusted
  - As much as if he or she were to modify the parent itself

Visibility

Introduction

Improving Readability

- Descriptive names plus hierarchical packages makes for very long statements

```
Messages.Queue.Diagnostics.Inject_Fault (
   Fault    => Messages.Queue.Diagnostics.CRC_Failure,
   Position => Messages.Queue.Front);
```

- Operators treated as functions defeat the purpose of overloading

```
Complex1 := Complex_Types."+" (Complex2, Complex3);
```

- Ada has mechanisms to simplify hierarchies

Operators and Primitives

- *Operators*
  - Constructs which behave generally like functions but which differ syntactically or semantically
  - Typically arithmetic, comparison, and logical
- **Primitive operation**
  - Predefined operations such as = and + etc.
  - Subprograms declared in the same package as the type and which operate on the type
  - Inherited or overridden subprograms
  - For `tagged` types, class-wide subprograms
  - Enumeration literals

"use" Clauses

use Clauses

- Provide direct visibility into packages' exported items
  - *Direct Visibility* - as if object was referenced from within package being used
- May still use expanded name

```
package Ada.Text_IO is
  procedure Put_Line(...);
  procedure New_Line(...);
  ...
end Ada.Text_IO;

with Ada.Text_IO;
procedure Hello is
  use Ada.Text_IO;
begin
  Put_Line("Hello World");
  New_Line(3);
  Ada.Text_IO.Put_Line ("Good bye");
end Hello;
```

use Clause Syntax

- May have several, like **with** clauses
- Can refer to any visible package (including nested packages)
- Syntax

```
use_package_clause ::= use package_name {, package_name}
```

- Can only **use** a package
  - Subprograms have no contents to **use**

use Clause Scope

- Applies to end of body, from first occurrence

```
package Pkg_A is
   Constant_A : constant := 123;
end Pkg_A;

package Pkg_B is
   Constant_B : constant := 987;
end Pkg_B;

with Pkg_A;
with Pkg_B;
use Pkg_A; -- everything in Pkg_A is now visible
package P is
   A  : Integer := Constant_A; -- legal
   B1 : Integer := Constant_B; -- illegal
   use Pkg_B; -- everything in Pkg_B is now visible
   B2 : Integer := Constant_B; -- legal
   function F return Integer;
end P;

package body P is
  -- all of Pkg_A and Pkg_B is visible here
  function F return Integer is (Constant_A + Constant_B);
end P;
```

No Meaning Changes

- A new **use** clause won't change a program's meaning!
- Any directly visible names still refer to the original entities

```
package D is
  T : Float;
end D;

with D;
procedure P is
  procedure Q is
    T, X : Float;
  begin
    ...
    declare
      use D;
    begin
      -- With or without the clause, "T" means Q.T
      X := T;
    end;
    ...
  end Q;
```

No Ambiguity Introduction

```
package D is
  V : Boolean;
end D;

package E is
  V : Integer;
end E;
with D, E;

procedure P is
  procedure Q is
    use D, E;
  begin
    -- to use V here, must specify D.V or E.V
    ...
  end Q;
begin
...
```

use Clauses and Child Units

- A clause for a child does **not** imply one for its parent
- A clause for a parent makes the child **directly** visible
    - Since children are 'inside' declarative region of parent

```
package Parent is
  P1 : Integer;
end Parent;

package Parent.Child is
  PC1 : Integer;
end Parent.Child;

with Parent;
with Parent.Child; use Parent.Child;
procedure Demo is
  D1 : Integer := Parent.P1;
  D2 : Integer := Parent.Child.PC1;
  use Parent;
  D3 : Integer := P1; -- illegal
  D4 : Integer := PC1;
  ...
```

use Clause and Implicit Declarations

- Visibility rules apply to implicit declarations too

```
package P is
  type Int is range Lower .. Upper;
  -- implicit declarations
  -- function "+"(Left, Right : Int) return Int;
  -- function "="(Left, Right : Int) return Boolean;
end P;

with P;
procedure Test is
  A, B, C : P.Int := some_value;
begin
  C := A + B; -- illegal reference to operator
  C := P."+" (A,B);
  declare
    use P;
  begin
    C := A + B; -- now legal
  end;
end Test;
```

"use type" Clauses

use type Clauses

- Syntax

```
use_type_clause ::= use type subtype_mark
                                         {, subtype_mark};
```

- Makes operators directly visible for specified type
  - Implicit and explicit operator function declarations
  - Only those that mention the type in the profile
    - Parameters and/or result type
- More specific alternative to `use` clauses
  - Especially useful when multiple `use` clauses introduce ambiguity

use type Clause Example

```
package P is
  type Int is range Lower .. Upper;
  -- implicit declarations
  -- function "+"(Left, Right : Int) return Int;
  -- function "="(Left, Right : Int) return Boolean;
end P;
with P;
procedure Test is
  A, B, C : P.Int := some_value;
  use type P.Int;
  D : Int; -- not legal
begin
  C := A + B; -- operator is visible
end Test;
```

use Type Clauses and Multiple Types

- One clause can make ops for several types visible
  - When multiple types are in the profiles
- No need for multiple clauses in that case

```
package P is
  type Miles_T is digits 6;
  type Hours_T is digits 6;
  type Speed_T is digits 6;
  -- "use type" on any of Miles_T, Hours_T, Speed_T
  -- makes operator visible
  function "/"(Left : Miles_T;
               Right : Hours_T)
               return Speed_T;
end P;
```

Multiple **use type** Clauses

- May be necessary
- Only those that mention the type in their profile are made visible

```
package P is
  type T1 is range 1 .. 10;
  type T2 is range 1 .. 10;
  -- implicit
  -- function "+"(Left : T2; Right : T2) return T2;
  type T3 is range 1 .. 10;
  -- explicit
  function "+"(Left : T1; Right : T2) return T3;
end P;

with P;
procedure UseType is
  X1 : P.T1;
  X2 : P.T2;
  X3 : P.T3;
  use type P.T1;
begin
  X3 := X1 + X2; -- operator visible because it uses T1
  X2 := X2 + X2; -- operator not visible
end UseType;
```

"use all type" Clauses

use all type Clauses

Ada 2012

- Makes all primitive operations for the type visible
  - Not just operators
  - Especially, subprograms that are not operators
- Still need a `use` clause for other entities
  - Typically exceptions

use all type Clause Example

Ada 2012

```
package Complex is
  type Number is private;
  function "+" (Left, Right : Number) return Number;
  procedure Make (C : out Number;
                  From_Real, From_Imag : Float);
...
with Complex;
use all type Complex.Number;
procedure Demo is
  A, B, C : Complex.Number;
  procedure Non_Primitive (X : Complex.Number) is null;
begin
  -- "use all type" makes these available
  Make (A, From_Real => 1.0, From_Imag => 0.0);
  Make (B, From_Real => 1.0, From_Imag => 0.0);
  C := A + B;
  -- but not this one
  Non_Primitive (0);
end Demo;
```

use all type v. **use type** Example

Ada 2012

```
with Complex;   use type Complex.Number;
procedure Demo is
   A, B, C : Complex.Number;
begin
   -- these are always allowed
   Complex.Make (A, From_Real => 1.0, From_Imag => 0.0);
   Complex.Make (B, From_Real => 1.0, From_Imag => 0.0);
   -- "use type" does not give access to primitive operations
   Make (A, 1.0, 0.0); -- Compile error here
   -- but does give access to operators
   C := A + B;
   declare
      -- but if we add "use all type" we get more visibility
      use all type Complex.Number;
   begin
      Make (A, 1.0, 0.0); -- Not a compile error
   end;
end Demo;
```

Renaming Entities

Three Positives Make a Negative

- Good Coding Practices ...
  - Descriptive names
  - Modularization
  - Subsystem hierarchies
- Can result in cumbersome references

```
-- use cosine rule to determine distance between two points,
-- given angle and distances between observer and 2 points
-- A**2 = B**2 + C**2 - 2*B*C*cos(angle)
Observation.Sides (Viewpoint_Types.Point1_Point2) :=
  Math_Utilities.Square_Root
    (Observation.Sides (Viewpoint_Types.Observer_Point1)**2 +
     Observation.Sides (Viewpoint_Types.Observer_Point2)**2 -
     2.0 * Observation.Sides (Viewpoint_Types.Observer_Point1) *
       Observation.Sides (Viewpoint_Types.Observer_Point2) *
       Math_Utilities.Trigonometry.Cosine
         (Observation.Vertices (Viewpoint_Types.Observer)));
```

Writing Readable Code - Part 1

- We could use **use** on package names to remove some dot-notation

```
-- use cosine rule to determine distance between two points, given angle
-- and distances between observer and 2 points A**2 = B**2 + C**2 -
-- 2*B*C*cos(angle)
Observation.Sides (Point1_Point2) :=
  Square_Root
    (Observation.Sides (Observer_Point1)**2 +
     Observation.Sides (Observer_Point2)**2 -
     2.0 * Observation.Sides (Observer_Point1) *
       Observation.Sides (Observer_Point2) *
       Cosine (Observation.Vertices (Observer)));
```

- But that only shortens the problem, not simplifies it
  - If there are multiple "use" clauses in scope:
    - Reviewer may have hard time finding the correct definition
    - Homographs may cause ambiguous reference errors
- We want the ability to refer to certain entities by another name (like an alias) with full read/write access (unlike temporary variables)

The **renames** Keyword

- Certain entities can be renamed within a declarative region
  - Packages

```
package Trig renames Math.Trigonometry
```

  - Objects (or elements of objects)

```
Angles : Viewpoint_Types.Vertices_Array_T
         renames Observation.Vertices;
Required_Angle : Viewpoint_Types.Vertices_T
         renames Viewpoint_Types.Observer;
```

  - Subprograms

```
function Sqrt (X : Base_Types.Float_T)
               return Base_Types.Float_T
               renames Math.Square_Root;
```

Writing Readable Code - Part 2

- With **renames** our complicated code example is easier to understand
  - Executable code is very close to the specification
  - Declarations as "glue" to the implementation details

```
begin
   package Math renames Math_Utilities;
   package Trig renames Math.Trigonometry;

   function Sqrt (X : Base_Types.Float_T) return Base_Types.Float_T
     renames Math.Square_Root;
   function Cos ...

   B : Base_Types.Float_T
     renames Observation.Sides (Viewpoint_Types.Observer_Point1);
   -- Rename the others as Side2, Angles, Required_Angle, Desired_Side
begin
   ...
   -- A**2 = B**2 + C**2 - 2*B*C*cos(angle)
   A := Sqrt (B**2 + C**2 - 2.0 * B * C * Cos (Angle));
end;
```

Lab

Visibility Lab

- Requirements
  - Create two types packages for two different shapes. Each package should have the following components:
    - `Number_of_Sides` - indicates how many sides in the shape
    - `Side_T` - numeric value for length
    - `Shape_T` - array of `Side_T` elements whose length is `Number_of_Sides`
  - Create a main program that will
    - Create an object of each `Shape_T`
    - Set the values for each element in `Shape_T`
    - Add all the elements in each object and print the total
- Hints
  - There are multiple ways to resolve this!

Visibility Lab Solution - Types

```
package Quads is

   Number_Of_Sides : constant Natural := 4;
   type Side_T is range 0 .. 1_000;
   type Shape_T is array (1 .. Number_Of_Sides) of Side_T;

end Quads;

package Triangles is

   Number_Of_Sides : constant Natural := 3;
   type Side_T is range 0 .. 1_000;
   type Shape_T is array (1 .. Number_Of_Sides) of Side_T;

end Triangles;
```

Visibility Lab Solution - Main #1

```
with Ada.Text_IO; use Ada.Text_IO;
with Quads;
with Triangles;
procedure Main1 is

   use type Quads.Side_T;
   Q_Sides : Natural renames Quads.Number_Of_Sides;
   Quad    : Quads.Shape_T := (1, 2, 3, 4);
   Quad_Total : Quads.Side_T := 0;

   use type Triangles.Side_T;
   T_Sides  : Natural renames Triangles.Number_Of_Sides;
   Triangle : Triangles.Shape_T := (1, 2, 3);
   Triangle_Total : Triangles.Side_T := 0;

begin

   for I in 1 .. Q_Sides loop
      Quad_Total := Quad_Total + Quad (I);
   end loop;
   Put_Line ("Quad: " & Quads.Side_T'Image (Quad_Total));

   for I in 1 .. T_Sides loop
      Triangle_Total := Triangle_Total + Triangle (I);
   end loop;
   Put_Line ("Triangle: " & Triangles.Side_T'Image (Triangle_Total));

end Main1;
```

Visibility Lab Solution - Main #2

```
with Ada.Text_IO; use Ada.Text_IO;
with Quads;       use Quads;
with Triangles;   use Triangles;
procedure Main2 is
   function Q_Image (S : Quads.Side_T) return String
      renames Quads.Side_T'Image;
   Quad : Quads.Shape_T := (1, 2, 3, 4);
   Quad_Total : Quads.Side_T := 0;

   function T_Image (S : Triangles.Side_T) return String
      renames Triangles.Side_T'Image;
   Triangle : Triangles.Shape_T := (1, 2, 3);
   Triangle_Total : Triangles.Side_T := 0;

begin

   for I in Quad'Range loop
      Quad_Total := Quad_Total + Quad (I);
   end loop;
   Put_Line ("Quad: " & Q_Image (Quad_Total));

   for I in Triangle'Range loop
      Triangle_Total := Triangle_Total + Triangle (I);
   end loop;
   Put_Line ("Triangle: " & T_Image (Triangle_Total));

end Main2;
```

Summary

Summary

Ada 2012

- `use` clauses are not evil but can be abused
  - Can make it difficult for others to understand code
- `use all type` clauses are more likely in practice than `use type` clauses
  - Only available in Ada 2012 and later
- `Renames` allow us to alias entities to make code easier to read
  - Subprogram renaming has many other uses, such as adding / removing default parameter values

Access Types

Introduction

Access Types Design

- Memory-addressed objects are called *access types*
- Objects are associated to *pools* of memory
  - With different allocation / deallocation policies
- Access objects are **guaranteed** to always be meaningful
  - In the absence of `Unchecked_Deallocation`
  - And if pool-specific

- Ada

```
type Integer_Pool_Access
  is access Integer;
P_A : Integer_Pool_Access
  := new Integer;

type Integer_General_Access
  is access all Integer;
G : aliased Integer;
G_A : Integer_General_Access := G'access;
```

- C++

```
int * P_C = malloc (sizeof (int));
int * P_CPP = new int;
int * G_C = &Some_Int;
```

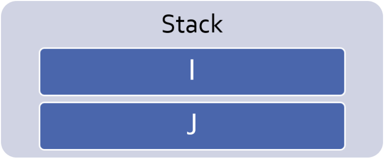
.

Access Types Can Be Dangerous

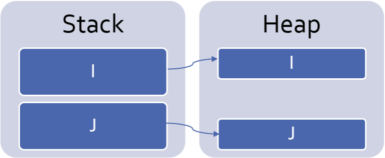
- Multiple memory issues
  - Leaks / corruptions
- Introduces potential random failures complicated to analyze
- Increase the complexity of the data structures
- May decrease the performances of the application
  - Dereferences are slightly more expensive than direct access
  - Allocations are a lot more expensive than stacking objects
- Ada avoids using accesses as much as possible
  - Arrays are not pointers
  - Parameters are implicitly passed by reference
- Only use them when needed

Stack vs Heap

```
I : Integer := 0;
J : String := "Some Long String";
```




```
I : Access_Int := new Integer'(0);
J : Access_Str := new String'("Some Long String");
```



Access Types

Declaration Location

- Can be at library level

```
package P is
  type String_Access is access String;
end P;
```

- Can be nested in a procedure

```
package body P is
   procedure Proc is
      type String_Access is access String;
   begin
      ...
   end Proc;
end P;
```

- Nesting adds non-trivial issues
  - Creates a nested pool with a nested accessibility
  - Don't do that unless you know what you are doing! (see later)

Null Values

- A pointer that does not point to any actual data has a **null** value
- Access types have a default value of **null**
- **null** can be used in assignments and comparisons

```
declare
   type Acc is access all Integer;
   V : Acc;
begin
   if V = null then
      --  will go here
   end if
   V := new Integer'(0);
   V := null; -- semantically correct, but memory leak
```

Access Types and Primitives

- Subprogram using an access type are primitive of the **access type**
  - **Not** the type of the accessed object

```
type A_T is access all T;
procedure Proc (V : A_T); -- Primitive of A_T, not T
```

- Primitive of the type can be created with the `access` mode
  - **Anonymous** access type

```
procedure Proc (V : access T); -- Primitive of T
```

Dereferencing Access Types

- `.all` does the access dereference
  - Lets you access the object pointed to by the pointer
- `.all` is optional for
  - Access on a component of an array
  - Access on a component of a record

Dereference Examples

```
type R is record
  F1, F2 : Integer;
end record;
type A_Int is access Integer;
type A_String is access all String;
type A_R is access R;
V_Int    : A_Int := new Integer;
V_String : A_String := new String'("abc");
V_R      : A_R := new R;

V_Int.all := 0;
V_String.all := "cde";
V_String (1) := 'z'; -- similar to V_String.all (1) := 'z';
V_R.all := (0, 0);
V_R.F1 := 1; -- similar to V_R.all.F1 := 1;
```

Pool-Specific Access Types

Pool-Specific Access Type

- An access type is a type

```
type T is [...]
type T_Access is access T;
V : T_Access := new T;
```

- Conversion is **not** possible between pool-specific access types

Allocations

- Objects are created with the **new** reserved word
- The created object must be constrained
  - The constraint is given during the allocation

    ```
    V : String_Access := new String (1 .. 10);
    ```

- The object can be created by copying an existing object - using a qualifier

  ```
  V : String_Access := new String'("This is a String");
  ```

Deallocations

- Deallocations are unsafe
  - Multiple deallocations problems
  - Memory corruptions
  - Access to deallocated objects
- As soon as you use them, you lose the safety of your access
- But sometimes, you have to do what you have to do ...
  - There's no simple way of doing it
  - Ada provides **Ada.Unchecked_Deallocation**
  - Has to be instantiated (it's a generic)
  - Must work on an object, reset to `null` afterwards

Deallocation Example

```
-- generic used to deallocate memory
with Ada.Unchecked_Deallocation;
procedure P is
   type An_Access is access A_Type;
   -- create instances of deallocation function
   -- (object type, access type)
   procedure Free is new Ada.Unchecked_Deallocation
     (A_Type, An_Access);
   V : An_Access := new A_Type;
begin
   Free (V);
   -- V is now null
end P;
```

General Access Types

General Access Types

- Can point to any pool (including stack)

```
type T is [...]
type T_Access is access all T;
V : T_Access := new T;
```

- Still distinct type
- Conversions are possible

```
type T_Access_2 is access all T;
V2 : T_Access_2 := T_Access_2 (V); -- legal
```

Referencing The Stack

- By default, stack-allocated objects cannot be referenced - and can even be optimized into a register by the compiler
- `aliased` declares an object to be referenceable through an access value

```
V : aliased Integer;
```

- `'Access` attribute gives a reference to the object

```
A : Int_Access := V'Access;
```

  - `'Unchecked_Access` does it **without checks**

Aliased Objects Examples

```
type Acc is access all Integer;
V, G : Acc;
I : aliased Integer;
...
V := I'Access;
V.all := 5; -- Same a I := 5
...
procedure P1 is
   I : aliased Integer;
begin
   G := I'Unchecked_Access;
   P2;
end P1;

procedure P2 is
begin
   -- OK when P2 called from P1.
   -- What if P2 is called from elsewhere?
   G.all := 5;
end P2;
```

Quiz

```
type One_T is access all Integer;
type Two_T is access Integer;

A : aliased Integer;
B : Integer;

One : One_T;
Two : Two_T;
```

Which assignment is legal?

A. `One := B'Access;`
B. `One := A'Access;`
C. `Two := B'Access;`
D. `Two := A'Access;`

Quiz

```
type One_T is access all Integer;
type Two_T is access Integer;

A : aliased Integer;
B : Integer;

One : One_T;
Two : Two_T;
```

Which assignment is legal?

A. `One := B'Access;`
B. *`One := A'Access;`*
C. `Two := B'Access;`
D. `Two := A'Access;`

`'Access` is only allowed for general access types (`One_T`). To use `'Access` on an object, the object must be **`aliased`**.

Accessibility Checks

Introduction to Accessibility Checks (1/2)

- The *depth* of an object depends on its nesting within declarative scopes

```
package body P is
   --  Library level, depth 0
   O0 : aliased Integer;
   procedure Proc is
      --  Library level subprogram, depth 1
      type Acc1 is access all Integer;
      procedure Nested is
         -- Nested subprogram, enclosing + 1, here 2
         O2 : aliased Integer;
```

- Objects can be referenced by access **types** that are at **same depth or deeper**
  - An **access scope** must be $\leq$ the object scope
- `type Acc1` (depth 1) can access `O0` (depth 0) but not **O2** (depth 2)
- The compiler checks it statically
  - Removing checks is a workaround!
- Note: Subprogram library units are at **depth 1** and not 0

Introduction to Accessibility Checks (2/2)

```
package body P is
   type T0 is access all Integer;
   A0 : T0;
   V0 : aliased Integer;
   procedure Proc is
      type T1 is access all Integer;
      A1 : T1;
      V1 : aliased Integer;
   begin
      A0 := V0'Access;
      A0 := V1'Access; -- illegal
      A0 := V1'Unchecked_Access;
      A1 := V0'Access;
      A1 := V1'Access;
      A1 := T1 (A0);
      A1 := new Integer;
      A0 := T0 (A1); -- illegal
  end Proc;
end P;
```

- To avoid having to face these issues, avoid nested access types

Getting Around Accessibility Checks

- Sometimes it is OK to use unsafe accesses to data
- `'Unchecked_Access` allows access to a variable of an incompatible accessibility level
- Beware of potential problems!

```
type Acc is access all Integer;
G : Acc;
procedure P is
   V : aliased Integer;
begin
   G := V'Unchecked_Access;
   ...
   Do_Something (G.all);
   G := null; -- This is "reasonable"
end P;
```

Using Access Types For Recursive Structures

- It is not possible to declare recursive structure
- But there can be an access to the enclosing type

```
type Cell; -- partial declaration
type Cell_Access is access all Cell;
type Cell is record -- full declaration
   Next       : Cell_Access;
   Some_Value : Integer;
end record;
```

Quiz

```
type Global_Access_T is access all Integer;
Global_Pointer : Global_Access_T;
Global_Object  : aliased Integer;
procedure Proc_Access is
   type Local_Access_T is access all Integer;
   Local_Pointer : Local_Access_T;
   Local_Object  : aliased Integer;
begin
```

Which assignment is **not** legal?

A. `Global_Pointer := Global_Object'Access;`
B. `Global_Pointer := Local_Object'Access;`
C. `Local_Pointer  := Global_Object'Access;`
D. `Local_Pointer  := Local_Object'Access;`

Quiz

```
type Global_Access_T is access all Integer;
Global_Pointer : Global_Access_T;
Global_Object  : aliased Integer;
procedure Proc_Access is
   type Local_Access_T is access all Integer;
   Local_Pointer : Local_Access_T;
   Local_Object  : aliased Integer;
begin
```

Which assignment is **not** legal?

A. `Global_Pointer := Global_Object'Access;`
B. *`Global_Pointer := Local_Object'Access;`*
C. `Local_Pointer  := Global_Object'Access;`
D. `Local_Pointer  := Local_Object'Access;`

Explanations

A. Pointer type has same depth as object
B. Pointer type is not allowed to have higher level than pointed-to object
C. Pointer type has lower depth than pointed-to object
D. Pointer type has same depth as object

Memory Management

Common Memory Problems (1/3)

- Uninitialized pointers

```
declare
   type An_Access is access all Integer;
   V : An_Access;
begin
   V.all := 5; -- constraint error
```

- Double deallocation

```
declare
   type An_Access is access all Integer;
   procedure Free is new
      Ada.Unchecked_Deallocation (Integer, An_Access);
   V1 : An_Access := new Integer;
   V2 : An_Access := V1;
begin
   Free (V1);
   ...
   Free (V2);
```

  - May raise `Storage_Error` if memory is still protected (unallocated)
  - May deallocate a different object if memory has been reallocated
    - Putting that object in an inconsistent state

Common Memory Problems (2/3)

- Accessing deallocated memory

```
declare
   type An_Access is access all Integer;
   procedure Free is new
      Ada.Unchecked_Deallocation (Integer, An_Access);
   V1 : An_Access := new Integer;
   V2 : An_Access := V1;
begin
   Free (V1);
   ...
   V2.all := 5;
```

  - May raise `Storage_Error` if memory is still protected (unallocated)
  - May modify a different object if memory has been reallocated (putting that object in an inconsistent state)

Common Memory Problems (3/3)

- Memory leaks

```
declare
   type An_Access is access all Integer;
   procedure Free is new
      Ada.Unchecked_Deallocation (Integer, An_Access);
   V : An_Access := new Integer;
begin
   V := null;
```

   - Silent problem
      - Might raise `Storage_Error` if too many leaks
      - Might slow down the program if too many page faults

How To Fix Memory Problems?

- There is no language-defined solution
- Use the debugger!
- Use additional tools
  - `gnatmem` monitor memory leaks
  - `valgrind` monitor all the dynamic memory
  - **GNAT.Debug_Pools** gives a pool for an access type, raising explicit exception in case of invalid access
  - Others...

Anonymous Access Types

Anonymous Access Parameters

- Parameter modes are of 4 types: `in`, `out`, `in out`, `access`
- The access mode is called *anonymous access type*
  - Anonymous access is implicitly general (no need for `all`)
- When used:
  - Any named access can be passed as parameter
  - Any anonymous access can be passed as parameter

```
type Acc is access all Integer;
Aliased_Integer : aliased Integer;
Access_Object   : Acc := Aliased_Integer'access;
procedure P1 (Anon_Access : access Integer) is null;
procedure P2 (Access_Parameter : access Integer) is
begin
   P1 (Aliased_Integer'access);
   P1 (Access_Object);
   P1 (Access_Parameter);
end P2;
```

Anonymous Access Types

- Other places can declare an anonymous access

```
function F return access Integer;
V : access Integer;
type T (V : access Integer) is record
  C : access Integer;
end record;
type A is array (Integer range <>) of access Integer;
```

- Do not use them without a clear understanding of accessibility check rules

Anonymous Access Constants

- `constant` (instead of `all`) denotes an access type through which the referenced object cannot be modified

```
type CAcc is access constant Integer;
G1 : aliased Integer;
G2 : aliased constant Integer := 123;
V1 : CAcc := G1'Access;
V2 : CAcc := G2'Access;
V1.all := 0; -- illegal
```

- `not null` denotes an access type for which null value cannot be accepted
  - Available in Ada 2005 and later

```
type NAcc is not null access Integer;
V : NAcc := null; -- illegal
```

- Also works for subprogram parameters

```
procedure Bar (V1 : access constant Integer);
procedure Foo (V1 : not null access Integer); -- Ada 2005
```

Lab

Access Types Lab

- Overview
  - Create a (really simple) Password Manager
    - The Password Manager should store the password and a counter for each of some number of logins
    - As it's a Password Manager, you want to modify the data directly (not pass the information around)
- Requirements
  - Create a Password Manager package
    - Create a record to store the password string and the counter
    - Create an array of these records indexed by the login identifier
    - The user should be able to retrieve a pointer to the record, either for modification or for viewing
  - Main program should:
    - Set passwords and initial counter values for many logins
    - Print password and counter value for each login
- Hint
  - Password is a string of varying length
    - Easiest way to do this is a pointer to a string that gets initialized to the correct length

Access Types Lab Solution - Password Manager

```
package Password_Manager is

   type Login_T is (Email, Banking, Amazon, Streaming);
   type Password_T is record
      Count    : Natural;
      Password : access String;
   end record;

   type Modifiable_T is access all Password_T;
   type Viewable_T is access constant Password_T;

   function Update (Login : Login_T) return Modifiable_T;
   function View (Login : Login_T) return Viewable_T;

end Password_Manager;

package body Password_Manager is

   Passwords : array (Login_T) of aliased Password_T;

   function Update (Login : Login_T) return Modifiable_T is
      (Passwords (Login)'Access);
   function View (Login : Login_T) return Viewable_T is
      (Passwords (Login)'Access);

end Password_Manager;
```

Access Types Lab Solution - Main

```
with Ada.Text_IO;      use Ada.Text_IO;
with Password_Manager; use Password_Manager;
procedure Main is

   procedure Update (Which : Password_Manager.Login_T;
                     Pw    : String;
                     Count : Natural) is
   begin
      Update (Which).Password := new String'(Pw);
      Update (Which).Count    := Count;
   end Update;

begin
   Update (Email, "QWE!@#", 1);
   Update (Banking, "asd123", 22);
   Update (Amazon, "098poi", 333);
   Update (Streaming, ")(*LKJ", 444);

   for Login in Login_T'Range loop
      Put_Line
        (Login'Image & " => " & View (Login).Password.all &
         View (Login).Count'Image);
   end loop;
end Main;
```

Summary

Summary

- Access types are the same as C/C++ pointers
- There are usually better ways of memory management
  - Language has its own ways of dealing with large objects passed as parameters
  - Language has libraries dedicated to memory allocation / deallocation
- At a minimum, create your own generics to do allocation / deallocation
  - Minimize memory leakage and corruption

Genericity

Introduction

The Notion of a Pattern

- Sometimes algorithms can be abstracted from types and subprograms

```
procedure Swap_Int (Left, Right : in out Integer) is
  V : Integer;
begin
   V := Left;
   Left := Right;
   Right := V;
end Swap_Int;

procedure Swap_Bool (Left, Right : in out Boolean) is
   V : Boolean;
begin
   V := Left;
   Left := Right;
   Right := V;
end Swap_Bool;
```

- It would be nice to extract these properties in some common pattern, and then just replace the parts that need to be replaced

```
-- T := Integer | Boolean
procedure Swap (Left, Right : in out T) is
  V : T;
begin
   V := Left;
   Left := Right;
   Right := V;
end Swap;
```

Solution: Generics

- A *generic unit* is a code pattern which can be reused
  - Does not get compiled as-is
- The instantiation applies the pattern to certain parameters
  - Based on properties
  - Use a *generic contract*
  - Parameters can be constant, variable, subprogram, type, package

Syntax

Usage

- Instantiated with the **new** keyword

```
--  Standard library
function Convert is new Ada.Unchecked_Conversion
  (Integer, Array_Of_4_Bytes);
--  Callbacks
procedure Parse_Tree is new Tree_Parser
  (Visitor_Procedure);
--  Containers, generic data-structures
package Integer_Stack is new Stack (Integer);
```

- Advanced usages for testing, proof, meta-programming

Declaration

- Subprograms

```
generic
   type T is private;
procedure Swap (L, R : in out T);
```

- Packages

```
generic
   type T is private;
package Stack is
   procedure Push (Item : T);
end Stack;
```

- Body is required
    - Will be specialized and compiled for **each instance**

Quiz

Which of the following statement is true?

A. Generics allow for code reuse
B. Generics can take packages as parameters
C. Genericity is specific to Ada
D. Genericity is available in all versions of Ada and/or SPARK

Quiz

Which of the following statement is true?

A. ***Generics allow for code reuse***
B. ***Generics can take packages as parameters***
C. Genericity is specific to Ada
D. ***Genericity is available in all versions of Ada and/or SPARK***

Quiz

Which one(s) of the following can be made generic?

```
generic
   type T is private;
<code goes here>
```

A. package
B. record
C. function
D. array

Quiz

Which one(s) of the following can be made generic?

```
generic
   type T is private;
<code goes here>
```

A. *package*
B. record
C. *function*
D. array

Only packages, functions, and procedures, can be made generic.

Quiz

Which of the following statement is true?

A. Generic instances must be nested inside a non-generic package
B. Generic instances must be created at compile-time
C. Generics instances can create new tagged types
D. Generics instances can create new tasks

Quiz

Which of the following statement is true?

A. Generic instances must be nested inside a non-generic package
B. Generic instances must be created at compile-time
C. ***Generics instances can create new tagged types***
D. ***Generics instances can create new tasks***

Generic instances can be created at any point, at a cost, and can do anything a package or subprogram can do, which make them versatile **but** potentially complex to use.

Generic Contracts

Definitions

- A formal generic parameter is a template
- Properties are either *constraints* or *capabilities*
  - Expressed from the **body** point of view
  - Constraints: e.g. unconstrained, `limited`
  - Capabilities: e.g. `tagged`, primitives

```
generic
   type Pv is private;           -- allocation, copy, assignment, "="
   with procedure Sort (T : Pv); -- primitive of Pv
   type Unc (<>) is private;     -- allocation require a value
   type Lim is limited private;  -- no copy or comparison
   type Disc is (<>);            -- 'First, ordering
package Generic_Pkg is [...]
```

- Actual parameter **may** require constraints, and **must** provide capabilities

```
package Pkg is new Generic_Pkg (
   Pv => Integer, -- has capabilities of private
   Sort => Sort -- procedure Sort (T : Integer)
   Unc => String,  -- uses "unconstrained" constraint
   Lim => Float,   -- does not use "limited" constraint
   Disc => Boolean, -- has capability of discrete
);
```

Syntax (partial)

```
type T1 is (<>); -- discrete
type T2 is range <>; -- Integer
type T3 is digits <>; -- float
type T4 is private; -- indefinite
type T5 (<>) is private; -- indefinite
type T6 is tagged;
type T7 is array (Boolean) of Integer;
type T8 is access Integer;
type T9 is limited private;
```

- Not limited to those choices

```
type T is not null access all limited tagged private;
```

Quiz

Which of the following statement is true?

A. Generic contracts define new types
B. Generic contracts can express any type constraint
C. Generic contracts can express inheritance constraint
D. Generic contracts can require a type to be numeric (`Real` or `Integer`)

Quiz

Which of the following statement is true?

A. Generic contracts define new types
B. Generic contracts can express any type constraint
C. ***Generic contracts can express inheritance constraint***
D. Generic contracts can require a type to be numeric (`Real` or `Integer`)

A. No, the formal type and the actual type just have different views
B. Counter-example: representation clauses

Quiz

```
generic
   type T1 is (<>);
   type T2 (<>) is private;
procedure Do_Something (A : T1; B : T2);
```

Which declaration(s) is(are) legal?

A. `procedure Do_A is new Do_Something (String, String)`
B. `procedure Do_B is new Do_Something (Character, Character)`
C. `procedure Do_C is new Do_Something (Integer, Integer)`
D. `procedure Do_D is new Do_Something (Boolean, Boolean)`

Quiz

```
generic
   type T1 is (<>);
   type T2 (<>) is private;
procedure Do_Something (A : T1; B : T2);
```

Which declaration(s) is(are) legal?

A. `procedure Do_A is new Do_Something (String, String)`
B. `procedure Do_B is new Do_Something (Character, Character)`
C. `procedure Do_C is new Do_Something (Integer, Integer)`
D. `procedure Do_D is new Do_Something (Boolean, Boolean)`

`T2` can be almost anything, so it's not the issue `T` must be discrete, so it cannot be `String`

Quiz

```
generic
   type T1 is (<>);
   type T2 (<>) is private;
procedure G
  (A : T1;
   B : T2);
```

Which is **not** a legal instantiation?

A. `procedure A is new G (String, Character);`
B. `procedure B is new G (Character, Integer);`
C. `procedure C is new G (Integer, Boolean);`
D. `procedure D is new G (Boolean, String);`

Quiz

```
generic
   type T1 is (<>);
   type T2 (<>) is private;
procedure G
  (A : T1;
   B : T2);
```

Which is **not** a legal instantiation?

A. *`procedure A is new G (String, Character);`*
B. `procedure B is new G (Character, Integer);`
C. `procedure C is new G (Integer, Boolean);`
D. `procedure D is new G (Boolean, String);`

T1 must be discrete - so an integer or an enumeration. T2 can be any type

Generic Formal Data

Generic Constants and Variables Parameters

- Variables can be specified on the generic contract
- The mode specifies the way the variable can be used:
  - **in** → read only
  - **in out** → read write
- Generic variables can be defined after generic types

```
generic
   type T is private;
   X1 : Integer;   -- constant
   X2 : in out T;  -- variable
procedure P;

V : Float;

procedure P_I is new P
   (T  => Float,
    X1 => 42,
    X2 => V);
```

Generic Subprogram Parameters

- Subprograms can be defined in the generic contract
- Must be introduced by **with** to differ from the generic unit

```
generic
   with procedure Callback;
procedure P;
procedure P is
begin
   Callback;
end P;
procedure Something;
procedure P_I is new P (Something);
```

Quiz

```
generic
   type Element_T is (<>);
   Last : in out Element_T;
procedure Write (P : Element_T);

Numeric        : Integer;
Enumerated     : Boolean;
Floating_Point : Float;
```

Which of the following piece(s) of code is(are) legal?

A. `procedure Write_A is new Write (Integer, Numeric)`
B. `procedure Write_B is new Write (Boolean, Enumerated)`
C. `procedure Write_C is new Write (Integer, Integer'Pos (Enumerated))`
D. `procedure Write_D is new Write (Float, Floating_Point)`

Quiz

```
generic
   type Element_T is (<>);
   Last : in out Element_T;
procedure Write (P : Element_T);

Numeric        : Integer;
Enumerated     : Boolean;
Floating_Point : Float;
```

Which of the following piece(s) of code is(are) legal?

A. *`procedure Write_A is new Write (Integer, Numeric)`*
B. *`procedure Write_B is new Write (Boolean, Enumerated)`*
C. `procedure Write_C is new Write (Integer, Integer'Pos (Enumerated))`
D. `procedure Write_D is new Write (Float, Floating_Point)`

A. Legal
B. Legal
C. The second generic parameter has to be a variable
D. The first generic parameter has to be discrete

Quiz

```
generic
   type L is limited private;
   type P is private;
procedure G_P;

type Lim is limited null record;
type Int is new Integer;

type Rec is record
   L : Lim;
   I : Int;
end record;
```

Which declaration(s) is(are) legal?

A. `procedure P is new G_P (Lim, Int)`
B. `procedure P is new G_P (Int, Rec)`
C. `procedure P is new G_P (Rec, Rec)`
D. `procedure P is new G_P (Int, Int)`

Quiz

```
generic
   type L is limited private;
   type P is private;
procedure G_P;

type Lim is limited null record;
type Int is new Integer;

type Rec is record
   L : Lim;
   I : Int;
end record;
```

Which declaration(s) is(are) legal?

A. *`procedure P is new G_P (Lim, Int)`*
B. `procedure P is new G_P (Int, Rec)`
C. `procedure P is new G_P (Rec, Rec)`
D. *`procedure P is new G_P (Int, Int)`*

Summary

Summary

- Generics are useful for **reusing code**
  - Sorting, containers, etc
- Generic contracts syntax is different from Ada declaration
  - But has some resemblance to it
  - e.g. discretes' `type Enum is (A, B, C)` vs generics' `type T is (<>)`
- Instantiation "generates" code
  - Costly
  - Beware of local generic instances!

Tagged Derivation

Introduction

Object-Oriented Programming With Tagged Types

- For `record` types

```
type T is tagged record
...
```

- Child types can add new components (*attributes*)
- Object of a child type can be **substituted** for base type
- Primitive (*method*) can *dispatch* **at runtime** depending on the type at call-site
- Types can be **extended** by other packages
  - Casting and qualification to base type is allowed
- Private data is encapsulated through **privacy**

Tagged Derivation Ada vs C++

```ada
type T1 is tagged record
  Member1 : Integer;
end record;

procedure Attr_F (This : T1);

type T2 is new T1 with record
  Member2 : Integer;
end record;

overriding procedure Attr_F (
     This : T2);
procedure Attr_F2 (This : T2);
```

```cpp
class T1 {
  public:
    int Member1;
    virtual void Attr_F(void);
  };

class T2 : public T1 {
  public:
    int Member2;
    virtual void Attr_F(void);
    virtual void Attr_F2(void)
  };
```

Tagged Derivation

Difference with Simple Derivation

- Tagged derivation **can** change the structure of a type
  - Keywords `tagged record` and `with record`

```
type Root is tagged record
   F1 : Integer;
end record;

type Child is new Root with record
   F2 : Integer;
end record;
```

Type Extension

- A tagged derivation **has** to be a type extension
    - Use `with null record` if there are no additional components

```
type Child is new Root with null record;
type Child is new Root; -- illegal
```

- Conversion is only allowed from **child to parent**

```
V1 : Root;
V2 : Child;
...
V1 := Root (V2);
V2 := Child (V1); -- illegal
```

Click here for more information on extending private types

Primitives

- Child **cannot remove** a primitive
- Child **can add** new primitives
- *Controlling parameter*
  - Parameters the subprogram is a primitive of
  - For `tagged` types, all should have the **same type**

```
type Root1 is tagged null record;
type Root2 is tagged null record;

procedure P1 (V1 : Root1;
              V2 : Root1);
procedure P2 (V1 : Root1;
              V2 : Root2); -- illegal
```

Freeze Point For Tagged Types

- Freeze point definition does not change
  - A variable of the type is declared
  - The type is derived
  - The end of the scope is reached
- Declaring tagged type primitives past freeze point is **forbidden**

```
type Root is tagged null record;

procedure Prim (V : Root);

type Child is new Root with null record; -- freeze root

procedure Prim2 (V : Root); -- illegal

V : Child; --  freeze child

procedure Prim3 (V : Child); -- illegal
```

Tagged Aggregate

- At initialization, all fields (including **inherited**) must have a **value**

```
type Root is tagged record
    F1 : Integer;
end record;

type Child is new Root with record
    F2 : Integer;
end record;

V : Child := (F1 => 0, F2 => 0);
```

- For **private types** use *aggregate extension*
  - Copy of a parent instance
  - Use `with null record` absent new fields

```
V2 : Child := (Parent_Instance with F2 => 0);
V3 : Empty_Child := (Parent_Instance with null record);
```

Click here for more information on aggregates of private extensions

Overriding Indicators

Ada 2005

- Optional **overriding** and **not overriding** indicators

```
type Shape_T is tagged record
   Name : String(1..10);
end record;

-- primitives of "Shape_T"
procedure Set_Name (S : in out Shape_T);
function Name (S : Shape_T) return string;

-- Derive "Point" from Shape_T
type Point is new Shape_T with record
   Origin : Coord_T;
end Point;

-- We want to _change_ the behavior of Set_Name
overriding procedure Set_Name (P : in out Point_T);
-- We want to _add_ a new primitive
not overriding Origin (P : Point_T) return Point_T;
-- We get "Name" for free
```

Prefix Notation

Ada 2012

- Tagged types primitives can be called as usual
- The call can use prefixed notation
  - **If** the first argument is a controlling parameter
  - No need for `use` or `use type` for visibility

```
-- Prim1 visible even without *use Pkg*
X.Prim1;

declare
   use Pkg;
begin
   Prim1 (X);
end;
```

Quiz

Which declaration(s) will make P a primitive of T1?

A.
```
type T1 is tagged null record;
procedure P (O : T1) is null;
```

B.
```
type T0 is tagged null record;
type T1 is new T0 with null record;
type T2 is new T0 with null record;
procedure P (O : T1) is null;
```

C.
```
type T1 is tagged null record;
Object : T1;
procedure P (O : T1) is null;
```

D.
```
package Nested is
  type T1 is tagged null record;
end Nested;
use Nested;
procedure P (O : T1) is null;
```

Quiz

Which declaration(s) will make P a primitive of T1?

A.
```
type T1 is tagged null record;
procedure P (O : T1) is null;
```

B.
```
type T0 is tagged null record;
type T1 is new T0 with null record;
type T2 is new T0 with null record;
procedure P (O : T1) is null;
```

C.
```
type T1 is tagged null record;
Object : T1;
procedure P (O : T1) is null;
```

D.
```
package Nested is
  type T1 is tagged null record;
end Nested;
use Nested;
procedure P (O : T1) is null;
```

A. Primitive (same scope)
B. Primitive (T1 is not yet frozen)
C. T1 is frozen by the object declaration
D. Primitive must be declared in same scope as type

Quiz

```
with Shapes; -- Defines tagged type Shape, with primitive P
with Colors; use Colors; -- Defines tagged type Color, with primitive P
with Weights; -- Defines tagged type Weight, with primitive P
use type Weights.Weight;

procedure Main is
   The_Shape : Shapes.Shape;
   The_Color : Colors.Color;
   The_Weight : Weights.Weight;
```

Which statement(s) is(are) valid?

A. `The_Shape.P`
B. `P (The_Shape)`
C. `P (The_Color)`
D. `P (The_Weight)`

Quiz

```
with Shapes; -- Defines tagged type Shape, with primitive P
with Colors; use Colors; -- Defines tagged type Color, with primitive P
with Weights; -- Defines tagged type Weight, with primitive P
use type Weights.Weight;

procedure Main is
   The_Shape : Shapes.Shape;
   The_Color : Colors.Color;
   The_Weight : Weights.Weight;
```

Which statement(s) is(are) valid?

A. *`The_Shape.P`*
B. `P (The_Shape)`
C. *`P (The_Color)`*
D. `P (The_Weight)`

D. `use type` only gives visibility to operators; needs to be `use all type`

Quiz

Which code block is legal?

A.
```
type A1 is record
  Field1 : Integer;
end record;
type A2 is new A1 with
null record;
```

B.
```
type B1 is tagged
record
  Field2 : Integer;
end record;
type B2 is new B1 with
record
  Field2b : Integer;
end record;
```

C.
```
type C1 is tagged
record
  Field3 : Integer;
end record;
type C2 is new C1 with
record
  Field3 : Integer;
end record;
```

D.
```
type D1 is tagged
record
  Field1 : Integer;
end record;
type D2 is new D1;
```

Quiz

Which code block is legal?

A.
```
type A1 is record
  Field1 : Integer;
end record;
type A2 is new A1 with
null record;
```

B.
```
type B1 is tagged
record
  Field2 : Integer;
end record;
type B2 is new B1 with
record
  Field2b : Integer;
end record;
```

C.
```
type C1 is tagged
record
  Field3 : Integer;
end record;
type C2 is new C1 with
record
  Field3 : Integer;
end record;
```

D.
```
type D1 is tagged
record
  Field1 : Integer;
end record;
type D2 is new D1;
```

Explanations

A. Cannot extend a non-tagged type
B. Correct
C. Components must have distinct names
D. Types derived from a tagged type must have an extension

Lab

Tagged Derivation Lab

- Requirements
  - Create a type structure that could be used in a business
    - A **person** has some defining characteristics
    - An **employee** is a *person* with some employment information
    - A **staff member** is an *employee* with specific job information
  - Create primitive operations to read and print the objects
  - Create a main program to test the objects and operations
- Hints
  - Use **overriding** and **not overriding** as appropriate **(Ada 2005 and above)**

Tagged Derivation Lab Solution - Types (Spec)

```
package Employee is
   subtype Name_T is String (1 .. 6);
   type Date_T is record
      Year  : Positive;
      Month : Positive;
      Day   : Positive;
   end record;
   type Job_T is (Sales, Engineer, Bookkeeping);

   ------------
   -- Person --
   ------------
   type Person_T is tagged record
      The_Name       : Name_T;
      The_Birth_Date : Date_T;
   end record;
   procedure Set_Name (O     : in out Person_T;
                       Value :        Name_T);
   function Name (O : Person_T) return Name_T;
   procedure Set_Birth_Date (O     : in out Person_T;
                             Value :        Date_T);
   function Birth_Date (O : Person_T) return Date_T;
   procedure Print (O : Person_T);

   --------------
   -- Employee --
   --------------
   type Employee_T is new Person_T with record
      The_Employee_Id : Positive;
      The_Start_Date  : Date_T;
   end record;
   not overriding procedure Set_Start_Date (O     : in out Employee_T;
                                            Value :        Date_T);
   not overriding function Start_Date (O : Employee_T) return Date_T;
   overriding procedure Print (O : Employee_T);

   --------------
   -- Position --
   --------------
   type Position_T is new Employee_T with record
      The_Job : Job_T;
   end record;
   not overriding procedure Set_Job (O     : in out Position_T;
                                     Value :        Job_T);
   not overriding function Job (O : Position_T) return Job_T;
   overriding procedure Print (O : Position_T);

end Employee;
```

Tagged Derivation Lab Solution - Types (Partial Body)

```
with Ada.Text_IO; use Ada.Text_IO;
package body Employee is

   function Image (Date : Date_T) return String is
     (Date.Year'Image & " -" & Date.Month'Image & " -" & Date.Day'Image);

   procedure Set_Name (O     : in out Person_T;
                       Value :        Name_T) is
   begin
      O.The_Name := Value;
   end Set_Name;
   function Name (O : Person_T) return Name_T is (O.The_Name);

   procedure Set_Birth_Date (O     : in out Person_T;
                             Value :        Date_T) is
   begin
      O.The_Birth_Date := Value;
   end Set_Birth_Date;
   function Birth_Date (O : Person_T) return Date_T is (O.The_Birth_Date);

   procedure Print (O : Person_T) is
   begin
      Put_Line ("Name: " & O.Name);
      Put_Line ("Birthdate: " & Image (O.Birth_Date));
   end Print;

   not overriding procedure Set_Start_Date
     (O     : in out Employee_T;
      Value :        Date_T) is
   begin
      O.The_Start_Date := Value;
   end Set_Start_Date;
   not overriding function Start_Date (O : Employee_T) return Date_T is
      (O.The_Start_Date);

   overriding procedure Print (O : Employee_T) is
   begin
      Put_Line ("Name: " & Name (O));
      Put_Line ("Birthdate: " & Image (O.Birth_Date));
      Put_Line ("Startdate: " & Image (O.Start_Date));
   end Print;

```

Tagged Derivation Lab Solution - Main

```
with Ada.Text_IO; use Ada.Text_IO;
with Employee;
procedure Main is
   Applicant : Employee.Person_T;
   Employ    : Employee.Employee_T;
   Staff     : Employee.Position_T;

begin
   Applicant.Set_Name ("Wilma ");
   Applicant.Set_Birth_Date ((Year  => 1_234,
                              Month => 12,
                              Day   => 1));

   Employ.Set_Name ("Betty ");
   Employ.Set_Birth_Date ((Year  => 2_345,
                           Month => 11,
                           Day   => 2));
   Employ.Set_Start_Date ((Year  => 3_456,
                           Month => 10,
                           Day   => 3));

   Staff.Set_Name ("Bambam");
   Staff.Set_Birth_Date ((Year  => 4_567,
                          Month => 9,
                          Day   => 4));
   Staff.Set_Start_Date ((Year  => 5_678,
                          Month => 8,
                          Day   => 5));
   Staff.Set_Job (Employee.Engineer);

   Applicant.Print;
   Employ.Print;
   Staff.Print;
end Main;
```

Summary

Summary

- Tagged derivation
  - Building block for OOP types in Ada
- Primitives rules for tagged types are trickier
  - Primitives **forbidden** below freeze point
  - **Unique** controlling parameter
  - Tip: Keep the number of tagged type per package low

Additional Information - Extending Tagged Types

How Do You Extend A Tagged Type?

- Premise of a tagged type is to *extend* an existing type
- In general, that means we want to add more fields
  - We can extend a `tagged` type by adding fields

```
package Animals is
  type Animal_T is tagged record
    Age : Natural;
  end record;
end Animals;

with Animals; use Animals;
package Mammals is
  type Mammal_T is new Animal_T with record
    Number_Of_Legs : Natural;
  end record;
end Mammals;

with Mammals; use Mammals;
package Canines is
  type Canine_T is new Mammal_T with record
    Domesticated : Boolean;
  end record;
end Canines;
```

Tagged Aggregates

- At initialization, all fields (including **inherited**) must have a **value**

```
Animal : Animal_T := (Age => 1);
Mammal : Mammal_T := (Age            => 2,
                      Number_Of_Legs => 2);
Canine : Canine_T := (Age            => 2,
                      Number_Of_Legs => 4,
                      Domesticated   => True);
```

- But we can also "seed" the aggregate with a parent object

```
Mammal := (Animal with Number_Of_Legs => 4);
Canine := (Animal with Number_Of_Legs => 4,
                       Domesticated   => False);
Canine := (Mammal with Domesticated => True);
```

Private Tagged Types

- But data hiding says types should be private!
- So we can define our base type as private

```
package Animals is
  type Animal_T is tagged private;
  function Get_Age (P : Animal_T) return Natural;
  procedure Set_Age (P : in out Animal_T; A : Natural);
private
  type Animal_T is tagged record
     Age : Natural;
  end record;
end Animals;
```

- And still allow derivation

```
with Animals;
package Mammals is
  type Mammal_T is new Animals.Animal_T with record
    Number_Of_Legs : Natural;
  end record;
```

- But now the only way to get access to `Age` is with accessor subprograms

Private Extensions

- In the previous slide, we exposed the fields for `Mammal_T`!
- Better would be to make the extension itself private

```
package Mammals is
  type Mammal_T is new Animals.Animal_T with private;
private
  type Mammal_T is new Animals.Animal_T with record
     Number_Of_Legs : Natural;
  end record;
end Mammals;
```

Click here to go back to Type Extension

Aggregates with Private Tagged Types

- Remember, an aggregate must specify values for all components
  - But with private types, we can't see all the components!
- So we need to use the "seed" method:

```
procedure Inside_Mammals_Pkg is
  Animal : Animal_T := Animals.Create;
  Mammal : Mammal_T;
begin
  Mammal := (Animal with Number_Of_Legs => 4);
  Mammal := (Animals.Create with Number_Of_Legs => 4);
end Inside_Mammals_Pkg;
```

- Note that we cannot use `others => <>` for components that are not visible to us

```
Mammal := (Number_Of_Legs => 4,
           others          => <>);  -- Compile Error
```

Null Extensions

- To create a new type with no additional fields
  - We still need to "extend" the record - we just do it with an empty record

```
type Dog_T is new Canine_T with null record;
```

- We still need to specify the "added" fields in an aggregate

```
C     : Canine_T := Canines.Create;
Dog1 : Dog_T := C; -- Compile Error
Dog2 : Dog_T := (C with null record);
```

Click here to go back to Tagged Aggregate

Quiz

Given the following code:

```
package Parents is
  type Parent_T is tagged private;
  function Create return Parent_T;
private
  type Parent_T is tagged record
     Id : Integer;
  end record;
end Parents;

with Parents; use Parents;
package Children is
  P : Parent_T;
  type Child_T is new Parent_T with record
     Count : Natural;
  end record;
  function Create (C : Natural) return Child_T;
end Children;
```

Which completion(s) of C is/are valid?

A. `function Create return Child_T is (Parents.Create with Count => 0);`
B. `function Create return Child_T is (others => <>);`
C. `function Create return Child_T is (0, 0);`
D. `function Create return Child_T is (P with Count => 0);`

Quiz

Given the following code:

```
package Parents is
  type Parent_T is tagged private;
  function Create return Parent_T;
private
  type Parent_T is tagged record
     Id : Integer;
  end record;
end Parents;

with Parents; use Parents;
package Children is
  P : Parent_T;
  type Child_T is new Parent_T with record
     Count : Natural;
  end record;
  function Create (C : Natural) return Child_T;
end Children;
```

Which completion(s) of C is/are valid?

A. *`function Create return Child_T is (Parents.Create with Count => 0);`*
B. `function Create return Child_T is (others => <>);`
C. `function Create return Child_T is (0, 0);`
D. *`function Create return Child_T is (P with Count => 0);`*

Explanations

A. Correct - `Parents.Create` returns `Parent_T`
B. Cannot use **others** to complete private part of an aggregate
C. Aggregate has no visibility to `Id` field, so cannot assign
D. Correct - `P` is a `Parent_T`

Low Level Programming

Introduction

Introduction

- Sometimes you need to get your hands dirty
- Hardware Issues
  - Register or memory access
  - Assembler code for speed or size issues
- Interfacing with other software
  - Object sizes
  - Endianness
  - Data conversion

Data Representation

Data Representation vs Requirements

- Developer usually defines requirements on a type

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

- The compiler then generates a representation for this type that can accommodate requirements
  - In GNAT, can be consulted using -gnatR2 switch

```
type My_Int is range 1 .. 10;
for My_Int'Object_Size use 8;
for My_Int'Value_Size  use 4;
for My_Int'Alignment   use 1;

-- using Ada 2012 aspects
type Ada2012_Int is range 1 .. 10
   with Object_Size => 8,
        Value_Size  => 4,
        Alignment   => 1;
```

- These values can be explicitly set, the compiler will check their consistency
- They can be queried as attributes if needed

```
X : Integer := My_Int'Alignment;
```

Value_Size / Size

- **Value_Size** (or **Size** in the Ada Reference Manual) is the minimal number of bits required to represent data
  - For example, `Boolean'Size = 1`
- The compiler is allowed to use larger size to represent an actual object, but will check that the minimal size is enough

```
type T1 is range 1 .. 4;
for T1'Size use 3;

-- using Ada 2012 aspects
type T2 is range 1 .. 4
   with Size => 3;
```

Object Size (GNAT-Specific)

- **Object_Size** represents the size of the object in memory
- It must be a multiple of **Alignment * Storage_Unit (8)**, and at least equal to **Size**

```
type T1 is range 1 .. 4;
for T1'Value_Size use 3;
for T1'Object_Size use 8;

-- using Ada 2012 aspects
type T2 is range 1 .. 4
   with Value_Size  => 3,
        Object_Size => 8;
```

- Object size is the *default* size of an object, can be changed if specific representations are given

Alignment

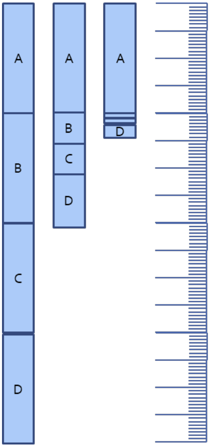
- Number of bytes on which the type has to be aligned
- Some alignment may be more efficient than others in terms of speed (e.g. boundaries of words (4, 8))
- Some alignment may be more efficient than others in terms of memory usage

```
type T1 is range 1 .. 4;
for T1'Size use 4;
for T1'Alignment use 8;

-- using Ada 2012 aspects
type T2 is range 1 .. 4
   with Size      => 4,
        Alignment => 8;
```

Record Types

- Ada doesn't force any particular memory layout
- Depending on optimization of constraints, layout can be optimized for speed, size, or not optimized

```
type Enum is (E1, E2, E3);
type Rec is record
   A : Integer;
   B : Boolean;
   C : Boolean;
   D : Enum;
end record;
```



Pack Aspect

- **pack** aspect (or pragma) applies to composite types (record and array)
- Compiler optimizes data for size no matter performance impact
- Unpacked

```
type Enum is (E1, E2, E3);
type Rec is record
   A : Integer;
   B : Boolean;
   C : Boolean;
   D : Enum;
end record;
type Ar is array (1 .. 1000) of Boolean;
-- Rec'Size is 56, Ar'Size is 8000
```

- Packed

```
type Enum is (E1, E2, E3);
type Rec is record
   A : Integer;
   B : Boolean;
   C : Boolean;
   D : Enum;
end record with Pack;
type Ar is array (1 .. 1000) of Boolean;
pragma Pack (Ar);
-- Rec'Size is 36, Ar'Size is 1000
```

Record Representation Clauses

- Exact mapping between a record and its binary representation
- Optimization purposes, or hardware requirements
  - Driver mapped on the address space, communication protocol...
- Fields represented as

```
<name> at <byte> range
    <starting-bit> ..
    <ending-bit>
```

```
type Rec1 is record
   A : Integer range 0 .. 4;
   B : Boolean;
   C : Integer;
   D : Enum;
end record;
for Rec1 use record
   A at 0 range 0 ..  2;
   B at 0 range 3 ..  3;
   C at 0 range 4 .. 35;
   -- unused space here
   D at 5 range 0 ..  2;
end record;
```

Array Representation Clauses

- `Component_Size` for array's **component's** size

```
type Ar1 is array (1 .. 1000) of Boolean;
for Ar1'Component_Size use 2;

-- using Ada 2012 aspects
type Ar2 is array (1 .. 1000) of Boolean
    with Component_Size => 2;
```

Endianness Specification

- **Bit_Order** for a type's endianness
- **Scalar_Storage_Order** for composite types
  - Endianess of components' ordering
  - GNAT-specific
  - Must be consistent with **Bit_Order**
- Compiler will peform needed bitwise transformations when performing operations

```
type Rec is record
   A : Integer;
   B : Boolean;
end record;
for Rec use record
   A at 0 range 0 .. 31;
   B at 0 range 32 .. 33;
end record;
for Rec'Bit_Order use System.High_Order_First;
for Rec'Scalar_Storage_Order use System.High_Order_First;

-- using Ada 2012 aspects
type Ar is array (1 .. 1000) of Boolean with
  Scalar_Storage_Order => System.Low_Order_First;
```

Change of Representation

- Explicit new type can be used to set representation
- Very useful to unpack data from file/hardware to speed up references

```
type Rec_T is record
     Field1 : Unsigned_8;
     Field2 : Unsigned_16;
     Field3 : Unsigned_8;
end record;
type Packed_Rec_T is new Rec_T;
for Packed_Rec_T use record
   Field1 at 0 range  0 ..  7;
   Field2 at 0 range  8 .. 23;
   Field3 at 0 range 24 .. 31;
end record;
R : Rec_T;
P : Packed_Rec_T;
...
R := Rec_T (P);
P := Packed_Rec_T (R);
```

Address Clauses and Overlays

Address

- Ada distinguishes the notions of
  - A reference to an object
  - An abstract notion of address (**System.Address**)
  - The integer representation of an address
- Safety is preserved by letting the developer manipulate the right level of abstraction
- Conversion between pointers, integers and addresses are possible
- The address of an object can be specified through the **Address** aspect

Address Clauses

- Ada allows specifying the address of an entity

```
Var : Unsigned_32;
for Var'Address use ... ;
```

- Very useful to declare I/O registers
  - For that purpose, the object should be declared volatile:

```
pragma Volatile (Var);
```

- Useful to read a value anywhere

```
function Get_Byte (Addr : Address) return Unsigned_8 is
  V : Unsigned_8;
  for V'Address use Addr;
  pragma Import (Ada, V);
begin
  return V;
end;
```

  - In particular the address doesn't need to be constant
  - But must match alignment

Address Values

- The type **Address** is declared in **System**
  - But this is a `private` type
  - You cannot use a number
- Ada standard way to set constant addresses:
  - Use **System.Storage_Elements** which allows arithmetic on address

```
for V'Address use
   System.Storage_Elements.To_Address (16#120#);
```

- GNAT specific attribute **'To_Address**
  - Handy but not portable

```
for V'Address use System'To_Address (16#120#);
```

Volatile

- The **Volatile** property can be set using an aspect (in Ada2012 only) or a pragma
- Ada also allows volatile types as well as objects

```
type Volatile_U16 is mod 2**16;
pragma Volatile(Volatile_U16);
type Volatile_U32 is mod 2**32 with Volatile; -- Ada 201
```

- The exact sequence of reads and writes from the source code must appear in the generated code
  - No optimization of reads and writes
- Volatile types are passed by-reference

Ada Address Example

```
type Bitfield is array (Integer range <>) of Boolean;
pragma Component_Size (1);

V  : aliased Integer; -- object can be referenced elsewhere
pragma Volatile (V);  -- may be updated at any time

V2 : aliased Integer;
pragma Volatile (V2);

V_A : System.Address := V'Address;
V_I : Integer_Address := To_Integer (V_A);

--  This maps directly on to the bits of V
V3 : aliased Bitfield (1 .. V'Size);
for V3'Address use V_A; -- overlay

V4 : aliased Integer;
--  Trust me, I know what I'm doing, this is V2
for V4'Address use To_Address (V_I - 4);
```

Aliasing Detection

- *Aliasing*: multiple objects are accessing the same address
  - Types can be different
  - Two pointers pointing to the same address
  - Two references onto the same address
  - Two objects at the same address
- `Var1'Has_Same_Storage (Var2)` checks if two objects occupy exactly the same space
- `Var'Overlaps_Storage (Var2)` checks if two object are partially or fully overlapping

Unchecked Conversion

- **Unchecked_Conversion** allows an unchecked *bitwise* conversion of data between two types
- Needs to be explicitly instantiated

```
type Bitfield is array (1 .. Integer'Size) of Boolean;
function To_Bitfield is new
   Ada.Unchecked_Conversion (Integer, Bitfield);
V : Integer;
V2 : Bitfield := To_Bitfield (V);
```

- Avoid conversion if the sizes don't match
    - Not defined by the standard
    - Many compilers will warn if the type sizes do not match

Inline Assembly

Calling Assembly Code

- Calling assembly code is a vendor-specific extension
- GNAT allows passing assembly with **System.Machine_Code.ASM**
  - Handled by the linker directly
- The developer is responsible for mapping variables on temporaries or registers
- See documentation
  - GNAT RM 13.1 Machine Code Insertion
  - GCC UG 6.39 Assembler Instructions with C Expression Operands

Simple Statement

- Instruction without inputs/outputs

```
Asm ("halt", Volatile => True);
```

  - You may specify **Volatile** to avoid compiler optimizations
  - In general, keep it False unless it created issues

- You can group several instructions

```
Asm ("nop" & ASCII.LF & ASCII.HT
     & "nop", Volatile => True);
Asm ("nop; nop", Volatile => True);
```

- The compiler doesn't check the assembly, only the assembler will
  - Error message might be difficult to read

Operands

- It is often useful to have inputs or outputs...
  - **Asm_Input** and **Asm_Output** attributes on types

```
Asm ("outb %0,%%dx",
     Inputs => (Unsigned_8'Asm_Input ("a", B),
                Port_Id'Asm_Input ("d", P)),
     Volatile => True);
```

First operand (starting with o)

Use %% for %

Input operand for type Unsigned_8

Constraint, 'a' for eax

Value

Instruction

Constraint, 'd' for edx

Name

Mapping Inputs / Outputs on Temporaries

```
Asm (<script referencing $<input> >,
     Inputs  => ({<type>'Asm_Input (<constraint>,
                                     <variable>)}),
     Outputs => ({<type>'Asm_Output (<constraint>,
                                      <variable>)});
```

- **assembly script** containing assembly instructions + references to registers and temporaries
- **constraint** specifies how variable can be mapped on memory (see documentation for full details)

Constraint	Meaning
R	General purpose register
M	Memory
F	Floating-point register
I	A constant
g	global (on x86)
a	eax (on x86)

Main Rules

- No control flow between assembler statements
  - Use Ada control flow statement
  - Or use control flow within one statement
- Avoid using fixed registers
  - Makes compiler's life more difficult
  - Let the compiler choose registers
  - You should correctly describe register constraints
- On x86, the assembler uses `AT&T` convention
  - First operand is source, second is destination
- See your toolchain's `as` assembler manual for syntax

Volatile and Clobber ASM Parameters

- `Volatile` → `True` deactivates optimizations with regards to suppressed instructions
- `Clobber` → `"reg1, reg2, ..."` contains the list of registers considered to be "destroyed" by the use of the ASM call
  - `memory` if the memory is accessed
    - Compiler won't use memory cache in registers across the instruction
  - `cc` if flags might have changed

Instruction Counter Example (x86)

```
with System.Machine_Code; use System.Machine_Code;
with Ada.Text_IO;         use Ada.Text_IO;
with Interfaces;          use Interfaces;
procedure Main is
   Low   : Unsigned_32;
   High  : Unsigned_32;
   Value : Unsigned_64;
   use ASCII;
begin
   Asm ("rdtsc" & LF,
        Outputs =>
           (Unsigned_32'Asm_Output ("=g", Low),
            Unsigned_32'Asm_Output ("=a", High)),
        Volatile => True);
   Values := Unsigned_64 (Low) +
             Unsigned_64 (High) * 2 ** 32;
   Put_Line (Values'Image);
end Main;
```

Reading a Machine Register (ppc)

```
function Get_MSR return MSR_Type is
   Res : MSR_Type;
begin
   Asm ("mfmsr %0",
        Outputs => MSR_Type'Asm_Output ("=r", Res),
        Volatile => True);
   return Res;
end Get_MSR;
generic
    Spr : Natural;
 function Get_Spr return Unsigned_32;
 function Get_Spr return Unsigned_32 is
    Res : Unsigned_32;
 begin
    Asm ("mfspr %0,%1",
         Inputs => Natural'Asm_Input ("K", Spr),
         Outputs => Unsigned_32'Asm_Output ("=r", Res),
         Volatile => True);
    return Res;
 end Get_Spr;
 function Get_Pir is new Get_Spr (286);
```

Writing a Machine Register (ppc)

```
generic
   Spr : Natural;
procedure Set_Spr (V : Unsigned_32);
procedure Set_Spr (V : Unsigned_32) is
begin
   Asm ("mtspr %0,%1",
        Inputs => (Natural'Asm_Input ("K", Spr),
                   Unsigned_32'Asm_Input ("r", V)));
end Set_Spr;
```

Tricks

Package Interfaces

- Package **Interfaces** provide Integer and unsigned types for many sizes
  - **Integer_8**, **Integer_16**, **Integer_32**, **Integer_64**
  - **Unsigned_8**, **Unsigned_16**, **Unsigned_32**, **Unsigned_64**
- With shift/rotation functions for unsigned types

Fat/Thin pointers for Arrays

- Unconstrained array access is a fat pointer

```
type String_Acc is access String;
Msg : String_Acc;
-- array bounds stored outside array pointer
```

- Use a size representation clause for a thin pointer

```
type String_Acc is access String;
for String_Acc'size use 32;
-- array bounds stored as part of array pointer
```

Flat Arrays

- A constrained array access is a thin pointer
  - No need to store bounds

```
type Line_Acc is access String (1 .. 80);
```

- You can use big flat array to index memory
  - See **GNAT.Table**
  - Not portable

```
type Char_array is array (natural) of Character;
type C_String_Acc is access Char_Array;
```

Lab

Low Level Programming Lab

(Simplified) Message generation / propagation

- Overview
    - Populate a message structure with data and a CRC (cyclic redundancy check)
    - "Send" and "Receive" messages and verify data is valid
- Goal
    - You should be able to create, "send", "receive", and print messages
    - Creation should include generation of a CRC to ensure data security
    - Receiving should include validation of CRC

Project Requirements

- Message Generation
  - Message should at least contain:
    - Unique Identifier
    - (Constrained) string field
    - Two other fields
    - CRC value
- "Send" / "Receive"
  - To simulate send/receive:
    - "Send" should do a byte-by-byte write to a text file
    - "Receive" should do a byte-by-byte read from that same text file
  - Receiver should validate received CRC is valid
    - You can edit the text file to corrupt data

Hints

- Use a representation clause to specify size of record
  - To get a valid size, individual components may need new types with their own rep spec
- CRC generation and file read/write should be similar processes
  - Need to convert a message into an array of "something"

Low Level Programming Lab Solution - CRC

```
with System;
package Crc is
   type Crc_T is mod 2**32;
   for Crc_T'size use 32;
   function Generate
     (Address : System.Address;
      Size : Natural)
      return Crc_T;
end Crc;

package body Crc is
   type Array_T is array (Positive range <>) of Crc_T;
   function Generate
     (Address : System.Address;
      Size    : Natural)
      return Crc_T is
      Word_Count : Natural;
      Retval     : Crc_T := 0;
   begin
      if Size > 0
      then
         Word_Count := Size / 32;
         if Word_Count * 32 /= Size
         then
            Word_Count := Word_Count + 1;
         end if;
         declare
            Overlay : Array_T (1 .. Word_Count);
            for Overlay'address use Address;
         begin
            for I in Overlay'range
            loop
               Retval := Retval + Overlay (I);
            end loop;
         end;
      end if;
      return Retval;
   end Generate;
end Crc;
```

Low Level Programming Lab Solution - Messages (Spec)

```
with Crc; use Crc;
package Messages is
   type Message_T is private;
   type Command_T is (Noop, Direction, Ascend, Descend, Speed);
   for Command_T use
     (Noop => 0, Direction => 1, Ascend => 2, Descend => 4, Speed => 8);
   for Command_T'size use 8;
   function Create (Command : Command_T;
                    Value   : Positive;
                    Text    : String := "")
                    return Message_T;
   function Get_Crc (Message : Message_T) return Crc_T;
   procedure Write (Message : Message_T);
   procedure Read (Message : out Message_T;
                   valid : out boolean);
   procedure Print (Message : Message_T);
private
   type U32_T is mod 2**32;
   for U32_T'size use 32;
   Max_Text_Length : constant := 20;
   type Text_Index_T is new Integer range 0 .. Max_Text_Length;
   for Text_Index_T'size use 8;
   type Text_T is record
      Text : String (1 .. Max_Text_Length);
      Last : Text_Index_T;
   end record;
   for Text_T'size use Max_Text_Length * 8 + Text_Index_T'size;
   type Message_T is record
      Unique_Id : U32_T;
      Command   : Command_T;
      Value     : U32_T;
      Text      : Text_T;
      Crc       : Crc_T;
   end record;
end Messages;
```

Low Level Programming Lab Solution - Main (Helpers)

```
with Ada.Text_IO; use Ada.Text_IO;
with Messages;
procedure Main is
   Message : Messages.Message_T;
   function Command return Messages.Command_T is
   begin
      loop
         Put ("Command (");
         for E in Messages.Command_T
         loop
            Put (Messages.Command_T'image (E) & " ");
         end loop;
         Put ("): ");
         begin
            return Messages.Command_T'value (Get_Line);
         exception
            when others =>
               Put_Line ("Illegal");
         end;
      end loop;
   end Command;
   function Value return Positive is
   begin
      loop
         Put ("Value: ");
         begin
            return Positive'value (Get_Line);
         exception
            when others =>
               Put_Line ("Illegal");
         end;
      end loop;
   end Value;
   function Text return String is
   begin
      Put ("Text: ");
      return Get_Line;
   end Text;
```

Low Level Programming Lab Solution - Main

```
   procedure Create is
      C : constant Messages.Command_T := Command;
      V : constant Positive           := Value;
      T : constant String             := Text;
   begin
      Message := Messages.Create
          (Command => C,
           Value   => V,
           Text    => T);
   end Create;
   procedure Read is
      Valid : Boolean;
   begin
      Messages.Read (Message, Valid);
      Ada.Text_IO.Put_Line("Message valid: " & Boolean'Image (Valid));
   end read;
begin
   loop
      Put ("Create Write Read Print: ");
      declare
         Command : constant String := Get_Line;
      begin
         exit when Command'length = 0;
         case Command (Command'first) is
            when 'c' | 'C' =>
               Create;
            when 'w' | 'W' =>
               Messages.Write (Message);
            when 'r' | 'R' =>
               read;
            when 'p' | 'P' =>
               Messages.Print (Message);
            when others =>
               null;
         end case;
      end;
   end loop;
end Main;
```

Low Level Programming Lab Solution - Messages (Helpers)

```
with Ada.Text_IO;
with Unchecked_Conversion;
package body Messages is
   Global_Unique_Id : U32_T := 0;
   function To_Text (Str : String) return Text_T is
      Length : Integer := Str'length;
      Retval : Text_T  := (Text => (others => ' '), Last => 0);
   begin
      if Str'length > Retval.Text'length then
         Length := Retval.Text'length;
      end if;
      Retval.Text (1 .. Length) := Str (Str'first .. Str'first + Length - 1);
      Retval.Last               := Text_Index_T (Length);
      return Retval;
   end To_Text;
   function From_Text (Text : Text_T) return String is
      Last : constant Integer := Integer (Text.Last);
   begin
      return Text.Text (1 .. Last);
   end From_Text;
   function Get_Crc (Message : Message_T) return Crc_T is
   begin
      return Message.Crc;
   end Get_Crc;
   function Validate (Original : Message_T) return Boolean is
      Clean : Message_T := Original;
   begin
      Clean.Crc := 0;
      return Crc.Generate (Clean'address, Clean'size) = Original.Crc;
   end Validate;
```

Low Level Programming Lab Solution - Messages (Body)

```
   function Create (Command : Command_T;
                    Value   : Positive;
                    Text    : String := "")
                    return Message_T is
      Retval : Message_T;
   begin
      Global_Unique_Id := Global_Unique_Id + 1;
      Retval           :=
        (Unique_Id => Global_Unique_Id, Command => Command,
         Value     => U32_T (Value), Text => To_Text (Text), Crc => 0);
      Retval.Crc := Crc.Generate (Retval'address, Retval'size);
      return Retval;
   end Create;
   type Char is new Character;
   for Char'size use 8;
   type Overlay_T is array (1 .. Message_T'size / 8) of Char;
   function Convert is new Unchecked_Conversion (Message_T, Overlay_T);
   function Convert is new Unchecked_Conversion (Overlay_T, Message_T);
   Const_Filename : constant String := "message.txt";
   procedure Write (Message : Message_T) is
      Overlay : constant Overlay_T := Convert (Message);
      File    : Ada.Text_IO.File_Type;
   begin
      Ada.Text_IO.Create (File, Ada.Text_IO.Out_File, Const_Filename);
      for I in Overlay'range loop
         Ada.Text_IO.Put (File, Character (Overlay (I)));
      end loop;
      Ada.Text_IO.New_Line (File);
      Ada.Text_IO.Close (File);
   end Write;
   procedure Read (Message : out Message_T;
                   Valid   : out Boolean) is
                   Overlay : Overlay_T;
                   File    : Ada.Text_IO.File_Type;
   begin
      Valid := False;
      Ada.Text_IO.Open (File, Ada.Text_IO.In_File, Const_Filename);
      declare
         Str : constant String := Ada.Text_IO.Get_Line (File);
      begin
         Ada.Text_IO.Close (File);
         for I in Str'range loop
            Overlay (I) := Char (Str (I));
         end loop;
         Message := Convert (Overlay);
         Valid   := Validate (Message);
      end;
   end Read;
   procedure Print (Message : Message_T) is
   begin
      Ada.Text_IO.Put_Line ("Message" & U32_T'image (Message.Unique_Id));
      Ada.Text_IO.Put_Line ("  " & Command_T'image (Message.Command) & " =>" &
                            U32_T'image (Message.Value));
      Ada.Text_IO.Put_Line ("  Additional Info: " & From_Text (Message.Text));
   end Print;
end Messages;
```

Summary

Summary

- Like C, Ada allows access to assembly-level programming
- Unlike C, Ada imposes some more restrictions to maintain some level of safety
- Ada also supplies language constructs and libraries to make low level programming easier

Ravenscar Tasking

Introduction

What Is the Ravenscar Profile?

- A **subset** of the Ada tasking model
  - Defined in the RM D.13
- Use concurrency in embedded real-time systems
  - Verifiable
  - Simple (Implemented reliably and efficiently)
- Scheduling theory for accurate analysis of real-time behavior
- Defined to help meet **safety-critical real-time** requirements
  - Determinism
  - Schedulability analysis
  - Memory-boundedness
  - Execution efficiency and small footprint
  - Certification
- `pragma Profile (Ravenscar)`

What Is the Jorvik profile?

- A **non-backwards compatible profile** based on Ravenscar
  - Defined in the RM D.13 (Ada 2022)
- Removes some constraints
  - Scheduling analysis may be harder to perform
- Subset of Ravenscar's requirements
- This class is about the more widespread Ravenscar
  - But some of Jorvik's differences are indicated
- `pragma Profile (Jorvik)`

What are GNAT runtimes?

- The *runtime* is an embedded library
  - Executing at run-time
  - In charge of standard's library support...
  - ...including tasking
- Standard runtime
  - Full runtime support
  - "Full-fledged" OS target (Linux, VxWorks...)
  - Large memory footprint
  - Full tasking (not shown in this class)
- Embedded runtime
  - Baremetal and RTOS targets
  - Reduced memory footprint
  - Most of runtime, except I/O and networking
  - Ravenscar / Jorvik tasking
- Light runtime
  - Baremetal targets
  - Very small memory footprint
  - Selected, very limited, runtime
  - Optional Ravenscar tasking (*Light-tasking* runtime)

A Simple Task

- Concurrent code execution via **task**
- `limited` types (No copies allowed)

```
package P is
   task type Put_T;

   T : Put_T;
end P;

package body P is
   task body Put_T is
   begin
      loop
         delay until Clock + Milliseconds (100);
         Put_Line ("T");
      end loop;
   end Put_T;
end P;
```

Two Ada Synchronization Models

- Passive
  - **Protected objects** model
  - Concurrency-safe **semantics**
- Active
  - Rendezvous
  - **Client** / **Server** model
- In Ravenscar: only **passive**

Tasks

Task Declaration

- Each instance of a task type is executing **concurrently**
- The **whole** tasking setup must be **static**
  - Compiler "compiles-in" the scheduling
- Task instances must be declared at the **library level**
  - Reminder: `main` declarative part is **not** at library level
- Body of a task must **never stop**
- Tasks should probably **yield**
  - For example with `delay until`
  - Or also a **protected entry guard** (see later)
  - Because of **Ravenscar scheduling** (see later)

Ravenscar Tasks Declaration Example

my_tasks.ads

```
package My_Tasks is
    task type Printer;

    P1 : Printer;
    P2 : Printer;
end My_Tasks;
```

my_tasks.adb

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

package body My_Tasks is
    P3 : Printer; --  correct

    task body Printer is
        Period : Time_Span := Milliseconds (100);
        Next : Time := Clock + Period;
        -- P : Printer -- /!\ Would be incorrect: not at library level
    begin
        loop
            Put_Line ("loops");

            --  Yielding
            delay until Next;
            Next := Next + Period;
        end loop;
    end Printer;
end My_Tasks;
```

Delays

Delay keyword

- `delay` keyword part of tasking
- Blocks for a time
- Absolute: Blocks until a given `Ada.Real_Time.Time`
- Relative: exists, but forbidden in Ravenscar

```
with Ada.Real_Time; use Ada.Real_Time;

procedure Main is
   Next : Time := Clock;
begin
   loop
      Next := Next + Milliseconds (10);
      delay until Next;
   end loop;
end Main;
```

Protected Objects

Protected Objects

- **Multitask-safe** accessors to get and set state
- **No** direct state manipulation
- **No** concurrent modifications
- `limited` types (No copies allowed)

```
protected type
  Protected_Value is
   procedure Set (V : Integer);
   function Get return Integer;
private
   Value : Integer;
end Protected_Value;
```

```
protected body Protected_Value is
   procedure Set (V : Integer) is
   begin
      Value := V;
   end Set;

   function Get return Integer is
   begin
      return Value;
   end Get;
end Protected_Value;
```

.

Misc: Single Declaration

- Instantiate an **anonymous** task (or protected) type
- Declares an object of that type
  - Body declaration is then using the **object** name

```
task Printer;

task body Printer is
begin
   loop
     Put_Line ("loops");
   end loop;
end Printer;
```

Protected: Functions and Procedures

- A `function` can **get** the state
  - Protected data is **read-only**
  - Concurrent call to `function` is **allowed**
  - **No** concurrent call to `procedure`
- A `procedure` can **set** the state
  - **No** concurrent call to either `procedure` or `function`
- In case of concurrency, other callers get **blocked**
  - Until call finishes

Protected entries

- A `entry` is equivalent to a procedure but
  - It can have a **guard condition**
    - Must be a **Boolean variable**
    - Declared as `private` member of the type
  - Calling task **blocks** on the guard until it is lifted
    - At most one task blocked (in Ravenscar)
  - At most one entry per protected type (in Ravenscar)

```
protected Blocker is
    entry Wait when Ready;
    procedure Mark_Ready; --  sets Ready to True
private
    Ready : Boolean := False;
end protected;
```

Ravenscar Scheduling

Ravenscar Patterns

- Periodic tasks (cyclic tasks / time triggered)
  - Sensor data acquisition
  - System monitoring
  - Control loops
  - Display update
- Event driven tasks
  - Alarm, Timeout
  - Interrupt
  - Data from another task
- Tasks can synchronize and communicate via protected objects

Task Activation

- Instantiated tasks start running when **activated**
- On the **stack**
  - When the **enclosing** package has finished **elaborating**
- Can be deferred to the end of **all** elaboration

my_tasks.ads

```
package My_Tasks is
   task type Foo_Task_T;

   T : Foo_Task_T;
   --  T is not running yet
end My_Tasks;
```

main.adb

```
with My_Tasks;
--  My_Tasks has finished elab, T runs

procedure Main is
[...]
```

Scheduling

- Priority-based
- No time slicing (quantum)
- A task executes until ...
    - The task is blocked
        - `delay until`
        - protected object `entry`
    - A higher priority task is woken up or unblocked (preemption)

Protected Objects and Interrupt Handling

- Simple protected operations
  - No queuing (except in Jorvik)
  - *Ceiling locking* on monoprocessor (see later)
  - *Proxy model* for protected entries
    - Entry body executed by the active task on behalf of the waiting tasks
    - Avoids unneeded context switches
    - Timing harder to analyze
- Simple, efficient, interrupt handling
  - Protected procedures as low level interrupt handlers
  - Procedure is *attached* to interrupt
  - Interrupt masking follows active priority

Some Advanced Concepts

Priorities

- Set by a `pragma Priority` or `Interrupt_Priority`
  - Can also use aspects
  - Tasks
  - Main subprogram (environment task)
  - `protected` definition
- Lower values mean lower priority
  - `Priority`
    - At least 30 levels
  - `Interrupt_Priority`
    - At least 1 level
    - > `Priority`

```
procedure Main is
  pragma Priority (2);

task T is
  pragma Priority (4);

protected Buffer is
   ...
private
   pragma Priority (3);
end Buffer;
```

Ceiling Locking

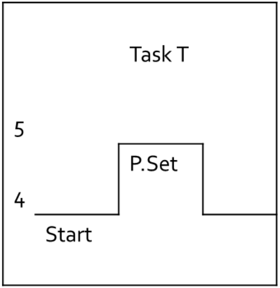
- Example of priority inversion:

```
L : Lock;

T1 : Task (Priority => 1);
T2 : Task (Priority => 2);
T3 : Task (Priority => 3);

T1 locks L
T3 starts, get scheduled (T3 > T1)
T3 tries to get L, blocks
T2 starts, get scheduled (T2 > T1)

Result: T2 running, T1 blocked, T3 blocked through L (but T3 > T2!)
```

- Solved with ceiling locking
    - Increase the priority of a task when it uses a protected
- Task priority is increased within a protected object
    - Condition: Task priority <= priorities of all protected objects it uses
    - Blocks other tasks without explicit locking
- `pragma Locking_Policy (Ceiling_Locking)`
    - Default on Ravenscar / Jorvik

Ceiling Locking Example

```
protected P with Priority => 5 is
   procedure Set (V : Integer);

task T with Priority => 4 is
  ...

task body T is
  ...
  P.Set (1);
```



Queue

- Protected `entry` are activated by **one** task at a time
- **Mutual exclusion** section
- Other tasks trying to enter
  - Are forbidden (Ravenscar)
  - Or are **queued** (Jorvik)
    - In **First-In First-Out** (FIFO) by default

Synchronous Task Control

- Primitives synchronization mechanisms and two-stage suspend operation
  - No critical section
  - More lightweight than protected objects
- Package exports a **Suspension_Object** type
  - Values are `True` and `False`, initially `False`
  - Such objects are awaited by (at most) one task
    - But can be set by several tasks

```
package Ada.Synchronous_Task_Control is
   type Suspension_Object is limited private;
   procedure Set_True (S : in out Suspension_Object);
   procedure Set_False (S : in out Suspension_Object);
   procedure Suspend_Until_True (S : in out Suspension_Object);
   function Current_State (S : Suspension_Object) return Boolean;
private
   ...
end Ada.Synchronous_Task_Control;
```

Timing Events

- User-defined actions executed at a specified wall-clock time
  - Calls back an `access protected procedure`
- Do not require a `task` or a `delay` statement

```
package Ada.Real_Time.Timing_Events is
   type Timing_Event is tagged 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);
   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
   ...
end Ada.Real_Time.Timing_Events;
```

Execution Time Clocks

- Not specific to Ravenscar / Jorvik
- Each task has an associated CPU time clock
    - Accessible via function call
- Clocks starts at creation time
    - **Before** activation
- Measures the task's total execution time
    - Including calls to libraries, OS services...
    - But not including time in a blocked or suspended state
- System and runtime also execute code
    - As well as interrupt handlers
    - Their execution time clock assignment is implementation-defined

Partition Elaboration Control

- Library units are elaborated in a partially-defined order
  - They can declare tasks and interrupt handlers
  - Once elaborated, tasks start executing
  - Interrupts may occur as soon as hardware is enabled
    - May be during elaboration
- This can cause race conditions
  - Not acceptable for certification
- `pragma Partition_Elaboration_Policy`

Partition Elaboration Policy

- `pragma Partition_Elaboration_Policy`
  - Defined in RM Annex H "High Integrity Systems"
- Controls tasks' activation
- Controls interrupts attachment
- Always relative to library units' elaboration
- **Concurrent policy**
  - Activation at the end of declaration's scope elaboration
  - Ada default policy
- **Sequential policy**
  - Deferred activation and attachment until **all** library units are activated
  - Easier scheduling analysis

Summary

Light-Tasking

- Everything is done by the Ada runtime
  - No OS underneath
- Simple
  - Less than 2800 Logical SLOCs
  - Footprint for simple tasking program is 10KB
- Static tasking model
  - Static tasks descriptors and stacks created at compile time
  - Task creation and activation is very simple
  - All tasks are created at initialization
- Simple protected operations
  - No queuing
  - Locking/unlocking by increasing/decreasing priority
- Complex features removed
  - Such as exception handling and propagation
- ECSS (E-ST-40C and Q-ST-80C) qualification material

Ada 2022

What's New

Types Syntax

- Image and litterals
  - `'Image` improvements
  - User-defined literals
- Composite Types
  - Improved aggregates
  - Iteration filters

Standard Lib

- `Ada.Numerics.Big_Numbers`
- `Ada.Strings.Text_Buffers`
- `System.Atomic_Operations`

Miscellaneous

- Jorvik profile
- Target name symbol
- Enumeration representation
- Staticness
- C variadics
- Subprogram access contracts
- Declare expression
- Simpler renames

Miscellaneous

Miscellaneous (1/2)

- Target Name Symbol (`@`)

```
Count := @ + 1;
```

- Enumeration representation attributes

```
type E is (A => 10, B => 20);
...
E'Enum_Rep (A); -- 10
E'Enum_Val (10); -- A
```

  - `'Enum_Rep` already present in GNAT

- Staticness

```
subtype T is Integer range 0 .. 2;
function In_T (A : Integer)
   return Boolean is
   (A in T) with Static;
```

- C variadic functions interface

```
procedure printf (format : String; opt_param : int)
   with Import, Convention => C_Variadic_1; -- Note the 1 for a single arg
```

Miscellaneous (2/2)

- Contract on access types

```
type A_F is access function (I : Integer) return Integer
   with Post => A_F'Result > I;
```

- Declare expressions

```
Area : Float := (declare
                    Pi : constant Float := 3.14159
                 begin
                    (Pi ** 2) * R);
```

- More expressive renamings

```
A : Integer;
B renames A; -- B type is infered
```

Image and Literals

Generalized 'Image

- All types have a `Image` attribute
- Its return value is (mostly) standardized
  - Except for e.g. unchecked unions
- Non-exhaustive example

- Code

```
Put_Line
   (Record_Obj'Image);
Put_Line
   (Array_Obj'Image);
Put_Line
   (Acc_O'Image);
Put_Line
   (Task_Obj'Image);
```

- Output

```
(I =>  1)
[
 (I =>  1),
 (I =>  1),
 (I =>  1),
 (I =>  1)]
(access 7ffd360de7f0)
(task task_obj_00000000240C0B0)
```

User-defined Image

- User-defined types can have a Image attribute
  - Need to specify the Put_Image aspect

```
type My_Type
[...]
  with Put_Image => My_Put_Image;
```

```
procedure My_Put_Image
  (Buffer : in out
            Ada.Strings.Text_Buffers.Root_Buffer_Type'Class;
   Arg    : in T);
```

- Using the new package Text_Buffers

User-defined 'Image example

- custom_image.ads

```
type R is null record with
  Put_Image => My_Put_Image;
```

- custom_image.adb

```
procedure My_Put_Image
  (Output : in out
     Ada.Strings.Text_Buffers.Root_Buffer_Type'Class;
   Obj    : R)
is
begin
   Output.Put ("my very own null record");
end My_Put_Image;
```

User-defined literals

- User-defined types can accept literals as inputs
  - `Integer`, `Float`, or `String`
  - Specifying a constructor to `Integer_Literal` aspect (resp `Float`, `String`)
- `my_int.ads`

```
type My_Int_T is private
   with Integer_Literal => Make_0;

function Make_0 (S : String) return My_Int_T;
...
type My_Int_T is record
    I : Integer;
end record;

function Make_0 (S : String) return My_Int_T is ((I => 0));
```

- `main.adb`

```
I : My_Int_T := 1;
```

Composite Types

Square Bracket Array Aggregates

- Only for **array** aggregates
    - **Required** in Ada 2022
    - **Forbidden** otherwise
    - Not backwards-compatible

```
A : array (1 .. 1) of Integer := [99]; -- Legal
B : array (1 .. 1) of Integer := (99); -- Not legal
```

- Allows for more complex initialization

```
O3 : A := [for I in 1 .. 10
           => (if I * I > 1 and I * I < 20 then I else 0)];
```

Iteration filters

- For any iteration
- Using the **when** keyword

```
for J in 1 .. 100 when J mod 2 /= 0 loop
```

- Can be used for aggregates as well

```
O4 : A := (for I of O3 when I /= 0 => I);
```

Container aggregates

- Using `with Aggregate => (<Args>)`
- Args are
  - `Empty` init function (or else default)
  - `Add_Named` named aggregate element
  - `Add_Unnamed` positional aggregate element
- You **cannot** mix named and unnamed

```
type JSON_Array is private
  with Aggregate => (Empty       => New_JSON_Array,
                     Add_Unnamed => Append);

function New_JSON_Array return JSON_Array;

procedure Append
   (Self  : in out JSON_Array;
    Value : JSON_Value) is null;

List : JSON.JSON_Array := [1, 2, 3];
```

- Implemented on standard lib's containers

Delta aggregates

- Can build an object from another one
  - Similarly to tagged types' extension aggregates
  - Using **with delta** in the aggregate

```
type Arr is array (1 .. 2) of Integer;
A : Arr := [3, 4];
B : Arr := [A with delta 1 => 0];

type Rec is record
   I1, I2 : Integer;
end record;
C : Rec := (I1 => 3, I2 => 4);
D : Rec := (C with delta I1 => 0);
```

Standard Lib

Ada.Numerics.Big_Numbers

- Numbers of arbitary size
  - Particularly useful for cryptography
- Big_Integers, Big_Reals child packages

```
type Big_Integer is private
  with Integer_Literal => From_Universal_Image,
       Put_Image => Put_Image;
subtype Big_Positive is Big_Integer [...]
subtype Big_Natural is Big_Integer [...]
subtype Valid_Big_Integer is [...]

function To_Big_Integer (Arg : Integer) return Valid_Big_Integer;
```

- Comparison operators

```
function "=" (L, R : Valid_Big_Integer) return Boolean;
function "<" (L, R : Valid_Big_Integer) return Boolean;
[...]
```

- Arithmetic operators

```
function "abs" (L : Valid_Big_Integer) return Valid_Big_Integer;
function "+" (L, R : Valid_Big_Integer) return Valid_Big_Integer;
[...]
```

Ada.Strings.Text_Buffers

- Object-oriented package
- Root_Buffer_Type
  - Basically a text stream
  - Abstract object

```
type Root_Buffer_Type is abstract tagged private [...];

procedure Put (
  Buffer : in out Root_Buffer_Type;
  Item   : in     String) is abstract;

procedure Wide_Put (
  Buffer : in out Root_Buffer_Type;
  Item   : in     Wide_String) is abstract;

procedure Wide_Wide_Put (
  Buffer : in out Root_Buffer_Type;
  Item   : in     Wide_Wide_String) is abstract;

procedure Put_UTF_8 (
  Buffer : in out Root_Buffer_Type;
  Item   : in     UTF_Encoding.UTF_8_String) is abstract;
```

System.Atomic_Operations

- Atomic types
  - May be used for lock-free synchronization
- Several child packages
  - `Exchange`
    - `function Atomic_Exchange` ...
  - `Test_And_Set`
    - `function Atomic_Test_And_Set` ...
  - `Integer_Arithmetic`, and `Modular_Arithmetic`
    - `generic package`
    - `procedure Atomic_Add` ...

Jorvik Profile

- A **non-backwards compatible profile** based on Ravenscar
  - Defined in the RM D.13 (Ada 2022)
- Remove some constraints
  - Number of protected entries, entry queue length...
  - Scheduling analysis may be harder to perform
- Subset of Ravenscars' requirements
- `pragma Profile (Jorvik)`

Summary

Ada 2022

- Adapting to new usages
    - Cryptography
    - Lock-free synchronizations
- More expressive syntax
    - `'Image` and literals
    - Functional approach: filters...
    - Simplified declarations and renamings
- Some features are not implemented...
    - ...by anyone
    - Those are related to parallelization
    - And are subject to future specification change

Unimplemented

- Global states
  - Available in SPARK
  - Declare side-effect in spec
- `parallel` reserved word
  - Parallelizes code
  - Conflict checking
  - Chunked iterators
  - Procedural iterators
    - `My_Map.Iterate (My_Procedure'Access)`

Ada Contracts

Introduction

Design-By-Contract

- Source code acting in roles of **client** and **supplier** under a binding **contract**
  - *Contract* specifies *requirements* or *guarantees*
    - *"A specification of a software element that affects its use by potential clients."* (Bertrand Meyer)
  - *Supplier* provides services
    - Guarantees specific functional behavior
    - Has requirements for guarantees to hold
  - *Client* utilizes services
    - Guarantees supplier's conditions are met
    - Requires result to follow the subprogram's guarantees

Ada Contracts

- Ada contracts include enforcement
  - At compile-time: specific constructs, features, and rules
  - At run-time: language-defined and user-defined exceptions
- Facilities prior to Ada 2012
  - Range specifications
  - Parameter modes
  - Generic contracts
  - OOP `interface` types (Ada 2005)
  - Work well, but on a restricted set of use-cases
- Contracts aspects are explicitly added in **Ada 2012**
  - Carried by subprograms
  - ... or by types (seen later)
  - Can have **arbitrary** conditions, more **versatile**

Assertion

- Boolean expression expected to be `True`
- Said *to hold* when `True`
- Language-defined **pragma**
  - The `Ada.Assertions.Assert` subprogram can wrap it

```
pragma Assert (not Full (Stack));
-- stack is not full
pragma Assert (Stack_Length = 0,
               Message => "stack was not empty");
-- stack is empty
```

- Raises language-defined `Assertion_Error` exception if expression does not hold

```
package Ada.Assertions is
  Assertion_Error : exception;
  procedure Assert (Check : in Boolean);
  procedure Assert (Check : in Boolean; Message : in String);
end Ada.Assertions;
```

Defensive Programming

- Should be replaced by subprogram contracts when possible

```
procedure Push (S : Stack) is
   Entry_Length : constant Positive := Length (S);
begin
   pragma Assert (not Is_Full (S)); -- entry condition
   [...]
   pragma Assert (Length (S) = Entry_Length + 1); -- exit condition
end Push;
```

- Subprogram contracts are an **assertion** mechanism
  - **Not** a drop-in replacement for all defensive code

```
procedure Force_Acquire (P : Peripheral) is
begin
   if not Available (P) then
      -- Corrective action
      Force_Release (P);
      pragma Assert (Available (P));
   end if;

   Acquire (P);
end;
```

Quiz

Which of the following statements is/are correct?

A. Contract principles apply only to Ada 2012
B. Contract should hold even for unique conditions and corner cases
C. Contract principles were first implemented in Ada
D. You cannot be both supplier and client

Quiz

Which of the following statements is/are correct?

A. Contract principles apply only to Ada 2012
B. ***Contract should hold even for unique conditions and corner cases***
C. Contract principles were first implemented in Ada
D. You cannot be both supplier and client

Explanations

A. No, but design-by-contract **aspects** are fully integrated to Ada 2012 design
B. Yes, special case should be included in the contract
C. No, in eiffel, in 1986!
D. No, in fact you are always **both**, even the `Main` has a caller!

Quiz

Which of the following statements is/are correct?

A. Assertions can be used in declarations
B. Assertions can be used in expressions
C. Any corrective action should happen before contract checks
D. Assertions must be checked using `pragma Assert`

Quiz

Which of the following statements is/are correct?

A. ***Assertions can be used in declarations***
B. Assertions can be used in expressions
C. ***Any corrective action should happen before contract checks***
D. Assertions must be checked using `pragma Assert`

Explanations

A. Will be checked at elaboration
B. No assertion expression, but `raise` expression exists
C. Exceptions as flow-control adds complexity, prefer a proactive `if` to a (reactive) `exception` handler
D. You can call `Ada.Assertions.Assert`, or even directly `raise Assertion_Error`

Quiz

Which of the following statements is/are correct?

A. Defensive coding is a good practice
B. Contracts can replace all defensive code
C. Contracts are executable constructs
D. Having exhaustive contracts will prevent runtime errors

Quiz

Which of the following statements is/are correct?

A. ***Defensive coding is a good practice***
B. Contracts can replace all defensive code
C. Contracts are executable constructs
D. Having exhaustive contracts will prevent runtime errors

Explanations

A. Principles are sane, contracts extend those
B. See previous slide example
C. e.g. generic contracts are resolved at compile-time
D. A failing contract **will cause** a runtime error, only extensive (dynamic / static) analysis of contracted code may provide confidence in the absence of runtime errors (AoRTE)

Preconditions and Postconditions

Subprogram-based Assertions

- **Explicit** part of a subprogram's **specification**
  - Unlike defensive code
- *Precondition*
  - Assertion expected to hold **prior to** subprogram call
- *Postcondition*
  - Assertion expected to hold **after** subprogram return
- Requirements and guarantees on both supplier and client
- Syntax uses **aspects**

```
procedure Push (This : in out Stack_T;
                Value : Content_T)
  with Pre  => not Full (This),
       Post => not Empty (This)
               and Top (This) = Value;
```

Requirements / Guarantees: Quiz

- Given the following piece of code

```
procedure Start is
begin
    ...
    Turn_On;
    ...

procedure Turn_On
 with Pre => Has_Power,
      Post => Is_On;
```

- Complete the table in terms of requirements and guarantees

	Client (Start)	Supplier (Turn_On)
Pre (Has_Power)		
Post (Is_On)		

Requirements / Guarantees: Quiz

- Given the following piece of code

```
procedure Start is
begin
    ...
    Turn_On;
    ...

procedure Turn_On
 with Pre => Has_Power,
      Post => Is_On;
```

- Complete the table in terms of requirements and guarantees

	Client (Start)	Supplier (Turn_On)
Pre (Has_Power)	Requirement	Guarantee
Post (Is_On)	Guarantee	Requirement

Examples

```
package Stack_Pkg is
   procedure Push (Item : in Integer) with
         Pre  => not Full,
         Post => not Empty and then Top = Item;
   procedure Pop (Item : out Integer) with
         Pre  => not Empty,
         Post => not Full;
   function Pop return Integer with
         Pre  => not Empty,
         Post => not Full;
   function Top return Integer with
         Pre => not Empty;
   function Empty return Boolean;
   function Full return Boolean;
end Stack_Pkg;

package body Stack_Pkg is
   Values  : array (1 .. 100) of Integer;
   Current : Natural := 0;

   -- Push/Pop cannot fail because preconditions prevent it
   procedure Push (Item : in Integer) is
   begin
      Current          := Current + 1;
      Values (Current) := Item;
   end Push;

   procedure Pop (Item : out Integer) is
   begin
      Item    := Values (Current);
      Current := Current - 1;
   end Pop;

   function Pop return Integer is
      Item : constant Integer := Values (Current);
   begin
      Current := Current - 1;
      return Item;
   end Pop;

   function Top return Integer is (Values (Current));
   function Empty return Boolean is (Current not in Values'Range);
   function Full return Boolean is (Current >= Values'Length);
end Stack_Pkg;
```

Preconditions

- Define obligations on client for successful call
  - Precondition specifies required conditions
  - Clients must meet precondition for supplier to succeed
- Boolean expressions
  - Arbitrary complexity
  - Specified via aspect name `Pre`
- Checked prior to call by client
  - `Assertion_Error` raised if false

```
procedure Push (This : in out Stack;  Value : Content)
  with Pre => not Full (This);
```

Postconditions

- Define obligations on supplier
  - Specify guaranteed conditions after call
- Boolean expressions (same as preconditions)
  - Specified via aspect name `Post`
- Content as for preconditions, plus some extras
- Checked after corresponding subprogram call
  - `Assertion_Error` raised if false

```
procedure Push (This : in out Stack;  Value : Content)
  with Pre  => not Full (This),
       Post => not Empty (This) and Top (This) = Value;
...
function Top (This : Stack) return Content
  with Pre => not Empty (This);
```

Postcondition 'Old Attribute

- Values as they were just before the call
- Uses language-defined attribute `'Old`
  - Can be applied to most any visible object
    - `limited` types are forbidden
    - May be expensive
  - Expression can be **arbitrary**
    - Typically `out`, `in out` parameters and globals

```
procedure Increment (This : in out Integer) with
    Pre  => This < Integer'Last,
    Post => This = This'Old + 1;
```

Function Postcondition `'Result` Attribute

- **`function`** result can be manipulated with `'Result`

Preconditions and Postconditions Example

- Multiple aspects separated by commas

```
procedure Push (This : in out Stack;
                Value : Content)
  with Pre  => not Full (This),
       Post => not Empty (This) and Top (This) = Value;
```

Quiz

```
function Area (L : Positive; H : Positive) return Positive is
   (L * H)
with Pre => ?
```

Which pre-condition is necessary for `Area` to calculate the correct result for all values `L` and `H`?

A. `L > 0 and H > 0`
B. `L < Positive'last and H < Positive'last`
C. `L * H in Positive`
D. None of the above

Quiz

```
function Area (L : Positive; H : Positive) return Positive is
   (L * H)
with Pre => ?
```

Which pre-condition is necessary for Area to calculate the correct result for all values L and H?

A. `L > 0 and H > 0`
B. `L < Positive'last and H < Positive'last`
C. `L * H in Positive`
D. ***None of the above***

Explanations

A. Parameters are `Positive`, so this is unnecessary
B. Overflow for large numbers
C. Classic trap: the check itself may cause an overflow!

The correct precondition would be

```
L > 0 and then H > 0 and then Integer'Last / L <= H
```

to prevent overflow errors on the range check.

Quiz

```
type Index_T is range 1 .. 100;
-- Database initialized such that value for element at I = I
Database : array (Index_T) of Integer;
-- Set the value for element Index to Value and
-- then increment Index by 1
function Set_And_Move (Value :        Integer;
                       Index : in out Index_T)
                       return Boolean
   with Post => ...
```

Given the following expressions, what is their value if they are evaluated in the postcondition of the call `Set_And_Move (-1, 10)`

```
Database'Old (Index)
Database (Index`Old)
Database (Index)'Old
```

Quiz

```
type Index_T is range 1 .. 100;
-- Database initialized such that value for element at I = I
Database : array (Index_T) of Integer;
-- Set the value for element Index to Value and
-- then increment Index by 1
function Set_And_Move (Value :        Integer;
                       Index : in out Index_T)
                       return Boolean
   with Post => ...
```

Given the following expressions, what is their value if they are evaluated in the postcondition of the call `Set_And_Move (-1, 10)`

`Database'Old (Index)`	11	Use new index in copy of original Database
``Database (Index`Old)``	-1	Use copy of original index in current Database
`Database (Index)'Old`	10	Evaluation of Database (Index) before call

Separations of Concerns

- Pre and Post fit together

```
function Val return Integer
 with Post => F'Result /= 0
is (if Val_Raw > 0 then Val_Raw else 1);

procedure Process (I : Integer)
 with Pre => I /= 0
is (Set_Output (10 / I));

[...]

Process (Val);
```

- Review of interface: guaranteed to work
  - What is returned by `Val` is always valid for `Process`
  - Need to check implementations
- Review of implementation
  - `Val` **always** returns a value that is `/= 0`
  - `Process` accepts **any** value that is `/= 0`
- Great separation of concerns
  - a team (Clients) could be in charge of reviewing the interface part
  - another team (Suppliers) could be in charge of reviewing the implementation part
  - both would use the contracts as a common understanding
  - Tools can do an automated review / validation: GNAT Static Analysis Suite, SPARK

No Secret Precondition Requirements

- Client should be able to **guarantee** them
- Enforced by the compiler

```
package P is
  function Foo return Bar
    with Pre => Hidden; -- illegal private reference
private
  function Hidden return Boolean;
end P;
```

Postconditions Are Good Documentation

```
procedure Reset
    (Unit : in out DMA_Controller;
     Stream : DMA_Stream_Selector)
  with Post =>
    not Enabled (Unit, Stream) and
    Operating_Mode (Unit, Stream) = Normal_Mode and
    Selected_Channel (Unit, Stream) = Channel_0 and
    not Double_Buffered (Unit, Stream) and
    Priority (Unit, Stream) = Priority_Low and
    (for all Interrupt in DMA_Interrupt =>
        not Interrupt_Enabled (Unit, Stream, Interrupt));
```

Contracts Code Reuse

- Contracts are about **usage** and **behaviour**
  - Not optimization
  - Not implementation details
  - **Abstraction** level is typically high
- Extracting them to `function` is a good idea
  - *Code as documentation, executable specification*
  - Completes the **interface** that the client has access to
  - Allows for **code reuse**

```
procedure Withdraw (This   : in out Account;
                    Amount :        Currency) with
  Pre  => Open (This) and Funds_Available (This, Amount),
  Post => Balance (This) = Balance (This)'Old - Amount;
...
function Funds_Available (This   : Account;
                          Amount : Currency)
                          return Boolean is
    (Amount > 0.0 and then Balance (This) >= Amount)
  with Pre => Open (This);
```

- A `function` may be unavoidable
  - Referencing private type components

Assertion Policy

- Assertions checks can be controled with `pragma Assertion_Policy`

```
pragma Assertion_Policy
     (Pre => Check,
      Post => Ignore);
```

- Fine **granularity** over assertion kinds and policy identifiers

https://docs.adacore.com/gnat_rm-docs/html/gnat_rm/gnat_rm/implementation_defined_pragmas.html#pragma-assertion-policy

- Certain advantage over explicit checks which are **harder** to disable
  - Conditional compilation via global `constant Boolean`

```
procedure Push (This : in out Stack;  Value : Content) is
begin
  if Debugging then
    if Full (This) then
      raise Overflow;
    end if;
  end if;
```

Type Invariants

Strong Typing

- Ada supports strong typing

```
type Small_Integer_T is range -1_000 .. 1_000;
type Enumerated_T is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
type Array_T is array (1 .. 3) of Boolean;
```

- What if we need stronger enforcement?
  - Number must be even
  - Subset of non-consecutive enumerals
  - Array should always be sorted
  - Type invariants are only checked on external boundaries
- **Type Invariant**
  - Property of type that is always true on **external** reference
  - *Guarantee* to client, similar to subprogram postcondition
- **Subtype Predicate**
  - Property of type that is always true, unconditionally
  - Can add arbitrary constraints to a type, unlike the "basic" type system

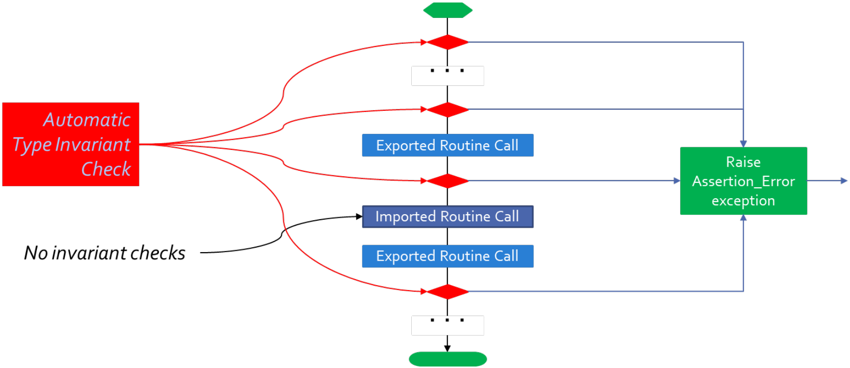
Examples

```
package Bank is
   type Account_T is private with Type_Invariant => Consistent_Balance (Account_T);
   type Currency_T is delta 0.01 digits 12;
   function Consistent_Balance (This : Account_T) return Boolean;
   procedure Open (This : in out Account_T; Initial_Deposit : Currency_T);
private
   type Vector_T is array (1 .. 100) of Currency_T;
   type Transaction_Vector_T is record
      Values : Vector_T;
      Count  : Natural := 0;
   end record;
   type Account_T is record -- initial state MUST satisfy invariant
      Current_Balance : Currency_T := 0.0;
      Withdrawals     : Transaction_Vector_T;
      Deposits        : Transaction_Vector_T;
   end record;
end Bank;

package body Bank is
   function Total (This : Transaction_Vector_T) return Currency_T is
      Result : Currency_T := 0.0;
   begin
      for I in 1 .. This.Count loop -- no iteration if list empty
         Result := Result + This.Values (I);
      end loop;
      return Result;
   end Total;
   function Consistent_Balance (This : Account_T) return Boolean is
      (Total (This.Deposits) - Total (This.Withdrawals) = This.Current_Balance);
   procedure Open (This : in out Account_T; Initial_Deposit : Currency_T) is
   begin
      This.Current_Balance := Initial_Deposit;
      -- if we checked, the invariant would be false here!
      This.Withdrawals.Count   := 0;
      This.Deposits.Count      := 1;
      This.Deposits.Values (1) := Initial_Deposit;
   end Open; -- invariant is now true
end Bank;
```

Type Invariant Verifications

- Automatically inserted by compiler
- Evaluated as postcondition of creation, evaluation, or return object
  - When objects first created
  - Assignment by clients
  - Type conversions
    - Creates new instances
- Not evaluated on internal state changes
  - Internal routine calls
  - Internal assignments
- Remember - these are abstract data types

Invariant Over Object Lifetime (Calls)



Example Type Invariant

- A bank account balance must always be consistent
  - Consistent Balance: Total Deposits - Total Withdrawals = Balance

```
package Bank is
  type Account is private with
    Type_Invariant => Consistent_Balance (Account);
  ...
  -- Called automatically for all Account objects
  function Consistent_Balance (This : Account)
    return Boolean;
  ...
private
  ...
end Bank;
```

Invariants Don't Apply Internally

- No checking within supplier package
  - Otherwise there would be no way to implement anything!
- Only matters when clients can observe state

```
procedure Open (This : in out Account;
                Name : in String;
                Initial_Deposit : in Currency) is
begin
  This.Owner := To_Unbounded_String (Name);
  This.Current_Balance := Initial_Deposit;
  -- invariant would be false here!
  This.Withdrawals := Transactions.Empty_Vector;
  This.Deposits := Transactions.Empty_Vector;
  This.Deposits.Append (Initial_Deposit);
  -- invariant is now true
end Open;
```

Quiz

```
package P is
   type Some_T is private;
   procedure Do_Something (X : in out Some_T);
private
   function Counter (I : Integer) return Boolean;
   type Some_T is new Integer with
      Type_Invariant => Counter (Integer (Some_T));
end P;

package body P is
   function Local_Do_Something (X : Some_T)
                                return Some_T is
      Z : Some_T := X + 1;
   begin
      return Z;
   end Local_Do_Something;
   procedure Do_Something (X : in out Some_T) is
   begin
      X := X + 1;
      X := Local_Do_Something (X);
   end Do_Something;
   function Counter (I : Integer)
                     return Boolean is
      (True);
end P;
```

If **Do_Something** is called from outside of P, how many times is **Counter** called?

A. 1
B. 2
C. 3
D. 4

Quiz

```
package P is
   type Some_T is private;
   procedure Do_Something (X : in out Some_T);
private
   function Counter (I : Integer) return Boolean;
   type Some_T is new Integer with
      Type_Invariant => Counter (Integer (Some_T));
end P;

package body P is
   function Local_Do_Something (X : Some_T)
                                return Some_T is
      Z : Some_T := X + 1;
   begin
      return Z;
   end Local_Do_Something;
   procedure Do_Something (X : in out Some_T) is
   begin
      X := X + 1;
      X := Local_Do_Something (X);
   end Do_Something;
   function Counter (I : Integer)
                     return Boolean is
      (True);
end P;
```

If **Do_Something** is called from outside of P, how many times is **Counter** called?

A. 1

B. *2*

C. 3

D. 4

Type Invariants are only evaluated on entry into and exit from externally visible subprograms. So `Counter` is called when entering and exiting `Do_Something` - not `Local_Do_Something`, even though a new instance of `Some_T` is created

Subtype Predicates

Examples

```
with Ada.Exceptions; use Ada.Exceptions;
with Ada.Text_IO;    use Ada.Text_IO;
procedure Predicates is

   subtype Even_T is Integer with Dynamic_Predicate => Even_T mod 2 = 0;
   type Serial_Baud_Rate_T is range 110 .. 115_200 with
      Static_Predicate => Serial_Baud_Rate_T in -- Non-contiguous range
         2_400 | 4_800 | 9_600 | 14_400 | 19_200 | 28_800 | 38_400 | 56_000;

   -- This must be dynamic because "others" will be evaluated at run-time
   subtype Vowel_T is Character with Dynamic_Predicate =>
        (case Vowel_T is when 'A' | 'E' | 'I' | 'O' | 'U' => True, when others => False);

   type Table_T is array (Integer range <>) of Integer;
   subtype Sorted_Table_T is Table_T (1 .. 5) with
        Dynamic_Predicate =>
        (for all K in Sorted_Table_T'Range =>
           (K = Sorted_Table_T'First or else Sorted_Table_T (K - 1) <= Sorted_Table_T (K)));

   J      : Even_T;
   Values : Sorted_Table_T := (1, 3, 5, 7, 9);

begin
   begin
      Put_Line ("J is" & J'Image);
      J := Integer'Succ (J); -- assertion failure here
      Put_Line ("J is" & J'Image);
      J := Integer'Succ (J); -- or maybe here
      Put_Line ("J is" & J'Image);
   exception
      when The_Err : others =>
         Put_Line (Exception_Message (The_Err));
   end;

   for Baud in Serial_Baud_Rate_T loop
      Put_Line (Baud'Image);
   end loop;

   Put_Line (Vowel_T'Image (Vowel_T'Succ ('A')));
   Put_Line (Vowel_T'Image (Vowel_T'Pred ('Z')));

   begin
      Values (3) := 0; -- not an exception
      Values     := (1, 3, 0, 7, 9); -- exception
   exception
      when The_Err : others =>
         Put_Line (Exception_Message (The_Err));
   end;
end Predicates;
```

Predicates

- Assertion expected to hold for all objects of given type
- Expressed as any legal boolean expression in Ada
  - Quantified and conditional expressions
  - Boolean function calls
- Two forms in Ada
  - **Static Predicates**
    - Specified via aspect named `Static_Predicate`
  - **Dynamic Predicates**
    - Specified via aspect named `Dynamic_Predicate`

type and subtype Predicates

- Applicable to both
- Applied via aspect clauses in both cases
- Syntax

```
type name is type_definition
   with aspect_mark [ => expression] { ,
             aspect_mark [ => expression] }

subtype defining_identifier is subtype_indication
   with aspect_mark [ => expression] { ,
             aspect_mark [ => expression] }
```

Why Two Predicate Forms?

	Static	Dynamic
Content	More Restricted	Less Restricted
Placement	Less Restricted	More Restricted

- Static predicates can be used in more contexts
  - More restrictions on content
  - Can be used in places Dynamic Predicates cannot
- Dynamic predicates have more expressive power
  - Fewer restrictions on content
  - Not as widely available

Subtype Predicate Examples

- Dynamic Predicate

```
subtype Even is Integer with Dynamic_Predicate =>
   Even mod 2 = 0; -- Boolean expression
   -- (Even indicates "current instance")
```

- Static Predicate

```
type Serial_Baud_Rate is range 110 .. 115200
  with Static_Predicate => Serial_Baud_Rate  in
    -- Non-contiguous range
    110  | 300  | 600 | 1200 | 2400 | 4800 |
    9600 | 14400 | 19200 | 28800 | 38400 | 56000 |
    57600 | 115200;
```

Predicate Checking

- Calls inserted automatically by compiler
- Violations raise exception `Assertion_Error`
    - When predicate does not hold (evaluates to False)
- Checks are done before value change
    - Same as language-defined constraint checks
- Associated variable is unchanged when violation is detected

Predicate Expression Content

- Reference to value of type itself, i.e., "current instance"

```
subtype Even is Integer
  with Dynamic_Predicate => Even mod 2 = 0;
J, K : Even := 42;
```

- Any visible object or function in scope
  - Does not have to be defined before use
  - Relaxation of "declared before referenced" rule of linear elaboration
  - Intended especially for (expression) functions declared in same package spec

Static Predicates

- *Static* means known at compile-time, informally
  - Language defines meaning formally (RM 3.2.4)
- Allowed in contexts in which compiler must be able to verify properties
- Content restrictions on predicate are necessary
- Ordinary Ada static expressions
- Static membership test selected by current instance
- Example

```
type Serial_Baud_Rate is range 110 .. 115200
  with Static_Predicate => Serial_Baud_Rate in
    -- Non-contiguous range
    110   | 300   | 600   | 1200  | 2400  | 4800  | 9600  |
    14400 | 19200 | 28800 | 38400 | 56000 | 57600 | 115200;
```

Dynamic Predicate Expression Content

- Any arbitrary boolean expression
  - Hence all allowed static predicates' content
- Plus additional operators, etc.

```
subtype Even is Integer
  with Dynamic_Predicate => Even mod 2 = 0;
subtype Vowel is Character with Dynamic_Predicate =>
  (case Vowel is
   when 'A' | 'E' | 'I' | 'O' | 'U' => True,
   when others => False); -- evaluated at run-time
```

- Plus calls to functions
  - User-defined
  - Language-defined

Beware Accidental Recursion In Predicate

- Involves functions because predicates are expressions
- Caused by checks on function arguments
- Infinitely recursive example

```
type Sorted_Table is array (1 .. N) of Integer with
   Dynamic_Predicate => Sorted (Sorted_Table);
-- on call, predicate is checked!
function Sorted (T : Sorted_Table) return Boolean;
```

- Non-recursive example

```
type Sorted_Table is array (1 .. N) of Integer with
   Dynamic_Predicate =>
   (for all K in Sorted_Table'Range =>
      (K = Sorted_Table'First
       or else Sorted_Table (K - 1) <= Sorted_Table (K)));
```

- Type-based example

```
type Table is array (1 .. N) of Integer;
subtype Sorted_Table is Table with
     Dynamic_Predicate => Sorted (Sorted_Table);
function Sorted (T : Table) return Boolean;
```

Quiz

```
type Days_T is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
function Is_Weekday (D : Days_T) return Boolean is
   (D /= Sun and then D /= Sat);
```

Which of the following is a valid subtype predicate?

A.
```
subtype T is Days_T with
   Static_Predicate => T in Sun | Sat;
```

B.
```
subtype T is Days_T with Static_Predicate =>
   (if T = Sun or else T = Sat then True else False);
```

C.
```
subtype T is Days_T with
   Static_Predicate => not Is_Weekday (T);
```

D.
```
subtype T is Days_T with
   Static_Predicate =>
     case T is when Sat | Sun => True,
               when others => False;
```

Quiz

```
type Days_T is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
function Is_Weekday (D : Days_T) return Boolean is
   (D /= Sun and then D /= Sat);
```

Which of the following is a valid subtype predicate?

A. *`subtype T is Days_T with Static_Predicate => T in Sun | Sat;`*

```
subtype T is Days_T with
   Static_Predicate => T in Sun | Sat;
```

B.
```
subtype T is Days_T with Static_Predicate =>
   (if T = Sun or else T = Sat then True else False);
```

C.
```
subtype T is Days_T with
   Static_Predicate => not Is_Weekday (T);
```

D.
```
subtype T is Days_T with
   Static_Predicate =>
     case T is when Sat | Sun => True,
               when others => False;
```

Explanations

A. Correct
B. `If` statement not allowed in a predicate
C. Function call not allowed in `Static_Predicate` (this would be OK for `Dynamic_Predicate`)
D. Missing parentheses around `case` expression

Summary

Working with Type Invariants

- They are not completely foolproof
  - External corruption is possible
  - Requires dubious usage
- Violations are intended to be supplier bugs
  - But not necessarily so, since not always bullet-proof
- However, reasonable designs will be foolproof

Type Invariants vs Predicates

- Type Invariants are valid at external boundary
  - Useful for complex types - type may not be consistent during an operation
- Predicates are like other constraint checks
  - Checked on declaration, assignment, calls, etc

Contract-Based Programming Benefits

- Facilitates building software with reliability built-in
  - Software cannot work well unless "well" is carefully defined
  - Clarifies design by defining obligations/benefits
- Enhances readability and understandability
  - Specification contains explicitly expressed properties of code
- Improves testability but also likelihood of passing!
- Aids in debugging
- Facilitates tool-based analysis
  - Compiler checks conformance to obligations
  - Static analyzers (e.g., SPARK, GNAT Static Analysis Suite) can verify explicit preconditions and postconditions

Annex - Ada Version Comparison

Ada Evolution

- Ada 83
  - Development late 70s
  - Adopted ANSI-MIL-STD-1815 Dec 10, 1980
  - Adopted ISO/8652-1987 Mar 12, 1987
- Ada 95
  - Early 90s
  - First ISO-standard OO language
- Ada 2005
  - Minor revision (amendment)
- Ada 2012
  - The new ISO standard of Ada

Programming Structure, Modularity

	Ada 83	Ada 95	Ada 2005	Ada 2012
Packages	✓	✓	✓	✓
Child units		✓	✓	✓
Limited with and mutually dependent specs			✓	✓
Generic units	✓	✓	✓	✓
Formal packages		✓	✓	✓
Partial parameterization			✓	✓
Conditional/Case expressions				✓
Quantified expressions				✓
In-out parameters for functions				✓
Iterators				✓
Expression functions				✓

Object-Oriented Programming

	Ada 83	Ada 95	Ada 2005	Ada 2012
Derived types	✓	✓	✓	✓
Tagged types		✓	✓	✓
Multiple inheritance of interfaces			✓	✓
Named access types	✓	✓	✓	✓
Access parameters, Access to subprograms		✓	✓	✓
Enhanced anonymous access types			✓	✓
Aggregates	✓	✓	✓	✓
Extension aggregates		✓	✓	✓
Aggregates of limited type			✓	✓
Unchecked deallocation	✓	✓	✓	✓
Controlled types, Accessibility rules		✓	✓	✓
Accessibility rules for anonymous types			✓	✓
Design-by-Contract aspects				✓

Concurrency

	Ada 83	Ada 95	Ada 2005	Ada 2012
Tasks	✓	✓	✓	✓
Protected types, Distributed annex		✓	✓	✓
Synchronized interfaces			✓	✓
Delays, Timed calls	✓	✓	✓	✓
Real-time annex		✓	✓	✓
Ravenscar profile, Scheduling policies			✓	✓
Multiprocessor affinity, barriers				✓
Re-queue on synchronized interfaces				✓
Ravenscar for multiprocessor systems				✓

Standard Libraries

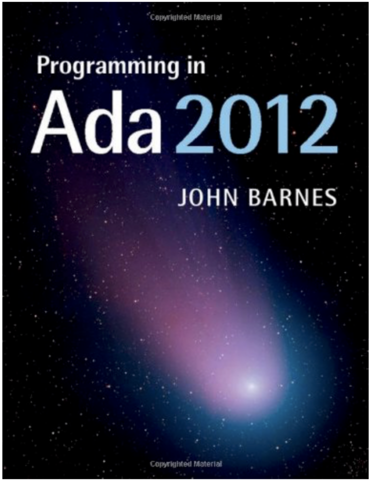
	Ada 83	Ada 95	Ada 2005	Ada 2012
Numeric types	✓	✓	✓	✓
Complex types		✓	✓	✓
Vector/matrix libraries			✓	✓
Input/output	✓	✓	✓	✓
Elementary functions		✓	✓	✓
Containers			✓	✓
Bounded Containers, holder containers, multiway trees				✓
Task-safe queues				✓
7-bit ASCII	✓	✓	✓	✓
8/16 bit		✓	✓	✓
8/16/32 bit (full Unicode)			✓	✓
String encoding package				✓

Annex - Reference Materials

General Ada Information

Learning the Ada Language

- Written as a tutorial for those new to Ada



Reference Manual

- **LRM** - Language Reference Manual (or just **RM**)
  - Always on-line (including all previous versions) at www.adaic.org
- Finding stuff in the RM
  - You will often see the RM cited like this **RM 4.5.3(10)**
  - This means *Section 4.5.3, paragraph 10*
  - Have a look at the table of contents
    - Knowing that chapter 5 is *Statements* is useful
  - Index is very long, but very good!

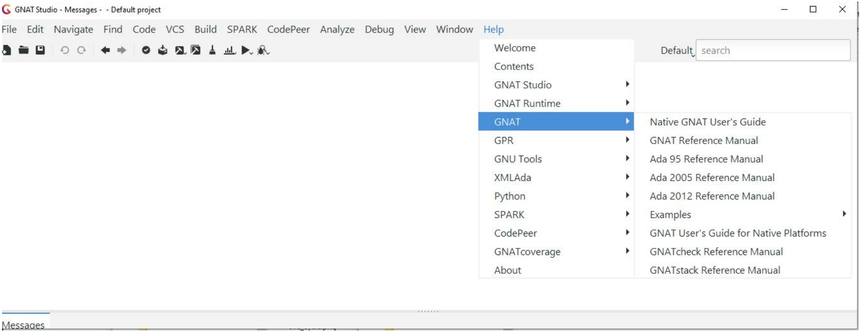
Current Ada Standard

- "ISO/IEC 8652(E) with Technical Corrigendum 1"
- Useful as a Reference Text but not intended to be read from beginning to end

GNAT-Specific Help

Reference Manual

- Reference Manual(s) available from GNAT STUDIO Help



GNAT Tools

- GNAT User's Guide
  - LOTS of info about the main tools: the GNAT compiler, binder, linker etc.
- GNAT Reference Manual
  - How GNAT implements Ada, pragmas, aspects, attributes etc. etc.
- GNAT STUDIO (the IDE)
  - Tutorial
  - User's Guide
  - Release notes
- Many other tools

AdaCore Support

Need More Help?

- If you have an AdaCore subscription:
  - Find out your customer number #XXXX
- Open a "TN" via the GNAT Tracker web interface and/or email
  - Send to: report@adacore.com
  - Subject should read: #XXXX - (descriptive text)
    - Where XXXX is your customer number
- Not just for "bug reports"
  - Ask questions, make suggestions etc. etc.