Introduction

About AdaCore

The Company

- Founded in 1994
- Centered around helping developers build **safe, secure and reliable** software
- Headquartered in New York and Paris
  - Representatives in countries around the globe
- Roots in Open Source software movement
  - GNAT compiler is part of GNU Compiler Collection (GCC)

About This Training

Your Trainer

- Experience in software development
  - Languages
  - Methodology
- Experience teaching this class

Goals of the training session

- What you should know by the end of the training
- Syllabus overview
  - The syllabus is a guide, but we might stray off of it
  - ...and that's OK: we're here to cover **your needs**

Roundtable

- 5 minute exercise
  - Write down your answers to the following
  - Then share it with the room
- Experience in software development
  - Languages
  - Methodology
- Experience and interest with the syllabus
  - Current and upcoming projects
  - Curious for something?
- Your personal goals for this training
  - What do you want to have coming out of this?
- Anecdotes, stories... feel free to share!
  - Most interesting or funny bug you've encountered?
  - Your own programming interests?

Course Presentation

- Slides
- Quizzes
- Labs
  - Hands-on practice
  - Recommended setup: latest GNAT Studio
  - Class reflection after some labs
- Demos
  - Depending on the context
- Daily schedule

Styles

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

> **Warning**
>
> This is a warning

> **Note**
>
> This is an important piece of info

> **Tip**
>
> This is a tip

Overview

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**
- International Standards Organization standard
  - Updated about every 10 years
- Writing **ADA** is like writing **CPLUSPLUS**

Ada Evolution Highlights

Ada 83 Abstract Data Types
Modules
Concurrency
Generics
Exceptions

Ada 95 OOP
Child Packages
Annexes

Ada 2005 Multiple Inheritance
Containers
Ravenscar

Ada 2012 Contracts
Iterators
Flexible Expressions

Ada 2022 `'Image` for all types
Declare expression

Note

Ada was created to be a **compiled**, **multi-paradigm** language with a **static** and **strong** type model

Big Picture

Language Structure (Ada95 and Onward)

- **Required** *Core* implementation
  - Always present in each compiler/run-time
    - Basic language contents (types, subprograms, packages, etc.)
    - Interface to Other Languages
- Optional *Specialized Needs Annexes*
  - No additional syntax
  - May be present or not depending on compiler/run-time
    - Real-Time Systems
    - Distributed Systems
    - Numerics
    - High-Integrity Systems

Core Language Content (1/3)

- Types
  - Language-defined types, including string
  - User-defined types
  - Static types keep things consistent
  - Strong types enforce constraints
- Subprograms
  - Syntax differs between *values* and *actions*
  - **`function`** for *value* and **`procedure`** for *action*
  - Overloading of names allowed
- Dynamic memory management
  - *access type* for abstraction of pointers
  - Access to static memory, allocated objects, subprograms
  - Accesses are **checked** (unless otherwise requested)
- Packages
  - Grouping of related entities
  - Separation of concerns
  - Information hiding

Core Language Content (2/3)

- Exceptions
  - Dealing with **errors**, **unexpected** events
  - Separate error-handling code from logic
- Generic Units
  - Code templates
  - Extensive parameterization for customization
- Object-Oriented Programming
  - Inheritance
  - Run-time polymorphism
  - Dynamic **dispatching**
- Contract-Based Programming
  - Pre- and post-conditions on subprograms
    - Formalizes specifications
  - Type invariants and predicates
    - Complex contracts on type definitions

Core Language Content (3/3)

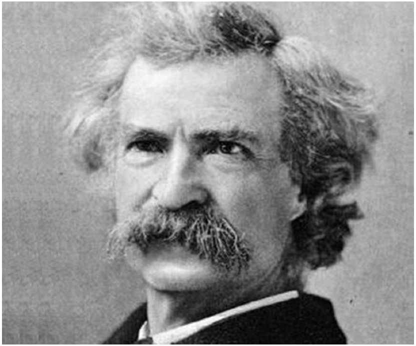
- Language-Based Concurrency
  - Explicit interactions
  - Run-time handling
  - Portable
- Low Level Programming
  - Define representation of types
  - Storage pools definition
  - Foreign language integration

Language Examination Summary

- 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

Ada Type Model

- Each *object* is associated a *type*
- **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

Declarations

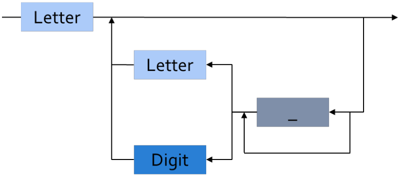
- *Declaration* associates a *name* to an *entity*
    - Objects
    - Types
    - Subprograms
    - et cetera
- In a *declarative part*
- Example: `N : Type := Value;`
    - `N` is usually an *identifier*

> ⚠ **Warning**
>
> Declaration **must precede** use

- **Some** implicit declarations
    - **Standard** types and operations
    - **Implementation**-defined

Identifiers and Comments

Identifiers



- Legal identifiers

  ```
  Phase2
  A
  Space_Person
  ```

- Not legal identifiers

  ```
  Phase2__1
  A_
  _space_person
  ```

- Character set **Unicode** 4.0
- Case **not significant**
  - **SpacePerson** $\Longleftrightarrow$ **SPACEPERSON**
  - ...but **different** from **Space_Person**

⚠ **Warning**

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
```

Declaring Constants / Variables (simplified)

- An *expression* is a piece of Ada code that returns a **value**.

```
<identifier> : constant := <expression>;
<identifier> : <type> := <expression>;
<identifier> : constant <type> := <expression>;
```

Quiz

Which statement(s) is (are) legal?

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

Quiz

Which statement(s) is (are) 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

Literals

String Literals

- A *literal* is a *textual* representation of a value in the code

```
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";
```

Decimal Numeric Literals

- Syntax

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

Tip

Underscore is **not** significant and used for grouping

- **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(s) is (are) 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(s) is (are) 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

Object Declarations

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

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

- Constant should have a value
  - Except for privacy (seen later)
- 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 facets:
  - **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(s) is (are) 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(s) is (are) 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` (not seen here)
- Match any integer / real type respectively
  - **Implicit** conversion, as needed

```
X : Integer64 := 2;
Y : Integer8 := 2;
F : Float := 2.0;
D : Long_Float := 2.0;
```

Numeric Literals Are Universally Typed

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

Tip

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

Tip
They have **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 **directly** 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

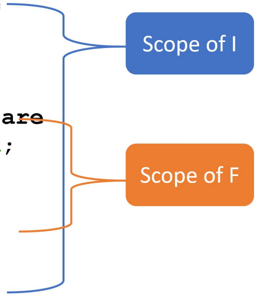
```
Swap: declare
  Temp : Integer;
begin
  Temp := U;
  U := V;
  V := Temp;
end Swap;
```

Scope and "Lifetime"

- Object in scope → exists

> ⚠ **Warning**
>
> 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;
```



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

Warning

- 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

Aspects

Pragmas

- Originated as a compiler directive for things like
  - Specifying the type of optimization

    ```
    pragma Optimize (Space);
    ```

  - Inlining of code

    ```
    pragma Inline (Some_Procedure);
    ```

  - Properties (*aspects*) of an entity
- Appearance in code
  - Unrecognized pragmas

    ```
    pragma My_Own_Pragma;
    ```

    - **No effect**
    - Cause **warning** (standard mode)
  - Must follow correct syntax

    ```
    pragma Page;             -- parameterless
    pragma Optimize (Off); -- with parameter
    ```

⚠ **Warning**

Malformed pragmas are **illegal**

```
pragma Illegal One;      -- compile error
```

Aspect Clauses

- 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**
  - **Recommended** over pragmas
- Syntax

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

Note

Aspect clauses always part of a **declaration**

Aspect Clause Example: Objects

- Updated **object syntax**

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

- Usage

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

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

Boolean Aspect Clauses

- **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

Strong Typing

- Definition of *type*
  - Applicable **values**
  - Applicable *primitive* **operations**
- Compiler-enforced
  - **Check** of values and operations
  - Easy for a computer

Tip

Developer can focus on **earlier** phase: requirement

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
```

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

Abstract Data Types (ADT)

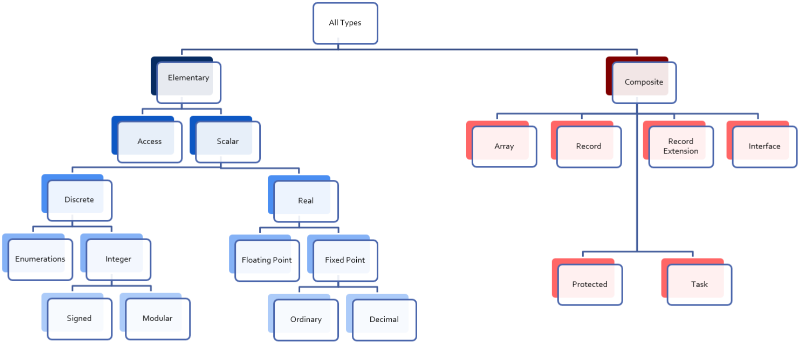
- **Variables** of the **type** encapsulate the **state**
- Classic definition of an ADT
  - Set of **values**
  - Set of **operations**
  - **Hidden** compile-time **representation**
- Compiler-enforced
  - Check of values and operation
  - Easy for a computer
  - Developer can focus on **earlier**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

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* (also known as *fields* or *elements*)
- **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

- Properties of entities that can be queried like a function
  - May take input parameters
- Defined by the language and/or compiler
  - Language-defined attributes found in RM K.2
  - *May* be implementation-defined
    - GNAT-defined attributes found in GNAT Reference Manual
  - Cannot be user-defined
- Attribute behavior is generally pre-defined
  - `Type_T'Digits` gives number of digits used in `Type_T` definition
- Some attributes can be modified by coding behavior
  - `Typemark'Size` gives the size of `Typemark`
    - Determined by compiler **OR** by using a representation clause
  - `Object'Image` gives a string representation of `Object`
    - Default behavior which can be replaced by aspect `Put_Image`
- Examples

```
J := Object'Size;
K := Array_Object'First(2);
```

Type Model Run-Time Costs

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

Note

Compile-time check is *free*

C

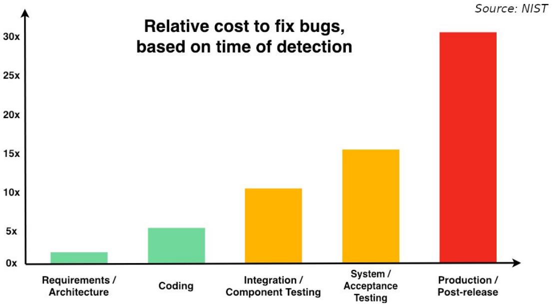
```c
int X;
int Y; // range 1 .. 10
...
if (X > 0 && X < 11)
  Y = X;
else
  // signal a failure
```

Ada

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

The Type Model Saves Money

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



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 := 0;
...
begin
   ...
   Count := Count + 1;
   ...
end;
```

Signed Integer Bounds

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

Predefined Signed 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**

> ⚠ **Warning**
>
> Portability not guaranteed
>
> - But may be difficult to avoid

Operators for Signed Integer Type

- By increasing precedence

relational operator = | /= | < | <= | > | >=
binary adding operator + | -
unary adding operator + | -
multiplying operator * | / | **mod** | **rem**
highest precedence operator ** | **abs**

Note

Exponentiation ** result will be a signed integer

- → power **must** be `Integer >= 0`

Warning

Division by zero → `Constraint_Error`

Signed 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
```

Signed 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

> **Note**
> 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 runtimes 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

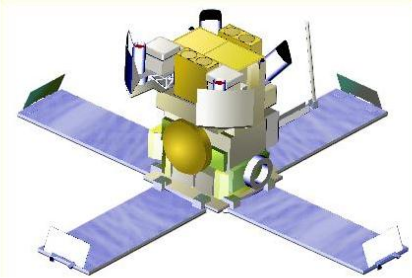
```
A : Boolean;
Counter : Integer;
...
A := (Counter = 22);
```

- 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(s) is (are) legal?

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(s) is (are) legal?

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 → 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**
- `System.Max_Digits` - constant specifying maximum digits of precision available for runtime

```
type Very_Precise_T is digits System.Max_Digits;
```

Need to do `with System;` *to get visibility*

Predefined Floating Point Types

- Type `Float` `>= 6` digits
- Additional implementation-defined types
  - `Long_Float` `>= 11` digits
- General-purpose

Tip

It is best, and easy, to **avoid** predefined types

- Loss of **portability**

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

⚠ Warning

`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. 7.6E-01
B. Compile Error
C. 8.0E-01
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. 7.6E-01
B. Compile Error
C. 8.0E-01
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
- Called as if it was a function
  - Named using destination type name

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

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

Default Value

- 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`

Note

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
- Constraints apply only to values
- Representation and set of operations are **kept**

Subtype Constraint Checks

- Constraints are checked
  - At initial value assignment
  - At assignment
  - At subprogram call
  - Upon return from subprograms
- Invalid constraints
  - Will cause `Constraint_Error` to be raised
  - May be detected at compile time
    - If values are **static**
    - Initial value → error
    - ... else → warning

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

Performance Impact of Constraints Checking

- Constraint checks have run-time performance impact
- The following code

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

- Generates assignment checks similar to

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

- These checks can be disabled with `-gnatp`

Optimizations of Constraint Checks

- Checks happen only if necessary
- Compiler assumes variables to be **initialized**
- So this code generates **no check**

```
procedure Demo is
  P, K : Integer range 0 .. 100;
begin
  P := K;
  --  But K is not initialized!
```

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

```
gnatstudio -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 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

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;
```

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
  - **null**
  - `A := B` (assignments)
  - **exit**
  - **goto**
  - **delay**
  - **raise**
  - `P (A, B)` (procedure calls)
  - **return**
  - Tasking-related: **requeue**, entry call `T.E (A, B)`, **abort**
- Compound
  - **if**
  - **case**
  - **loop** (and variants)
  - **declare**
  - Tasking-related: **accept**, **select**

Tasking-related are seen in the tasking chapter

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

```
declare
   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
begin
   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
```

Aliasing the Assignment Target

Ada 2022

- C allows you to simplify assignments when the target is used in the expression. This avoids duplicating (possibly long) names.

```
total = total + value;
// becomes
total += value;
```

- Ada 2022 implements this by using the target name symbol `@`

```
Total := Total + Value;
-- becomes
Total := @ + Value;
```

- Benefit
  - Symbol can be used multiple times in expression

```
Value := (if @ > 0 then @ else -(@));
```

- Limitation
  - Symbol is read-only (so it can't change during evaluation)

```
function Update (X : in out Integer) return Integer;
function Increment (X: Integer) return Integer;
```

```
13 Value := Update (@);
14 Value := Increment (@);
```

```
example.adb:13:21: error: actual for "X" must be a
variable
```

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(s) is (are) 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(s) is (are) 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 - No such "+" operator: must convert operand individually

D. Legal - Correct conversion and types

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 Direction 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 for an **if** block

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
  - 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(s) is (are) 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(s) is (are) 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

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

- We use **in** to indicate membership in a range of values

```
if Color in Red .. Green then
if Index in List'Range then
```

- But what if the values are not contiguous?
  - We could use a Boolean conjunction

    ```
    if Index = 1 or Index = 3 or Index = 5 then
    ```

  - Or we could simplify it by specifying a collection (or set)

    ```
    if Index in 1 | 3 | 5 then
    ```

    - | is used to separate members
    - So `1 | 3 | 5` is the set for which we are verifying membership

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(s) is (are) 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(s) is (are) 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. `Wed` and `Fri` are not Boolean expressions - need to compare each of them to `Today`
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 -- run-time 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

- 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

- 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

- 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(s) is (are) 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(s) is (are) 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

Array Types

Introduction

What Is an Array?

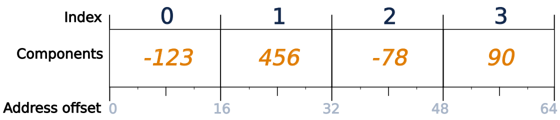
- Definition: collection of components of the same type, stored in contiguous memory, and indexed using a discrete range
- Syntax (simplified):

```
type <typename> is array (Index_Type) of Component_Type;
```

where

- *Index_Type*
  - Discrete range of values to be used to access the array components
- *Component_Type*
  - Type of values stored in the array
  - All components are of this same type and size

```
type Array_T is array (0 .. 3) of Interfaces.Integer_32;
```



Arrays in Ada

- 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;
```

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; -- run-time 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 (simplified)

```
type <typename> is array (<index constraint>) of <constrained type>;
```

where

typename *- identifier*

index constraint *- discrete range or type*

constrained type *- type with size known at compile time*

Examples

```
type Integer_Array_T is array (1 .. 3) of Integer;
type Boolean_Array_T is array (Boolean) of Integer;
type Character_Array_T is array (character range 'a' .. 'z') of Boolean;
type Copycat_T is array (Boolean_Array_T'Range) of Integer;
```

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(s) is (are) 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(s) is (are) 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

```
type Days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
type Schedule is array (Days range <>) of Float;
```

- Bounds set by:
  - Object declaration

    ```
    Weekdays : Schedule(Mon..Fri);
    ```

  - Object (or constant) initialization

    ```
    Weekend : Schedule := (Sat => 4.0, Sun => 0.0);
    -- (Note this is an array aggregate, explained later)
    ```

  - Further type definitions (shown later)
  - Actual parameter to subprogram (shown later)
- Once set, bounds never change

```
Weekdays(Sat) := 0.0; --  Constraint error
Weekend(Mon)  := 0.0; --  Constraint error
```

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);  -- run-time 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;
--  Index_T'Size = 8

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 Bit_T is range 0 .. 1;
type Bit_Array_T is array (Positive range <>) of Bit_T;
```

Which declaration(s) is (are) legal?

A. `AAA : Array_T (0..99);`
B. `BBB : Array_T (1..32);`
C. `CCC : Array_T (17..16);`
D. `DDD : Array_T;`

Quiz

```
type Bit_T is range 0 .. 1;
type Bit_Array_T is array (Positive range <>) of Bit_T;
```

Which declaration(s) is (are) legal?

A. `AAA : Array_T (0..99);`
B. *`BBB : Array_T (1..32);`*
C. *`CCC : Array_T (17..16);`*
D. `DDD : Array_T;`

Explanations

A. `Array_T` index is `Positive` which starts at 1
B. OK, indices are in range
C. OK, indicates a zero-length array
D. Object must be constrained

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 run-time 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
  - 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)

```
declare
   type Index1_T is range 1 .. 2;
   type Index2_T is range 101 .. 102;
   type Array1_T is array (Index1_T) of Integer;
   type Array2_T is array (Index2_T) of Integer;
   type Array3_T is array (Boolean) of Integer;

   One   : Array1_T;
   Two   : Array2_T;
   Three : Array3_T;

begin

   One := Array1_T (Two);     -- OK
   Two := Array2_T (Three);   -- Illegal (indices not convertible)
```

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!`

Example: Slicing with Explicit Indexes

- Imagine a requirement to have a ISO date
    - Year, month, and day with a specific format

```
declare
   Iso_Date : String (1 .. 10) := "2024-03-27";
begin
   Put_Line (Iso_Date);
   Put_Line (Iso_Date (1 .. 4));  --  year
   Put_Line (Iso_Date (6 .. 7));  --  month
   Put_Line (Iso_Date (9 .. 10)); --  day
```

Idiom: 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 Iso_Index is Positive range 1 .. 10;
  subtype Year is Iso_Index
     range Iso_Index'First .. Iso_Index'First + 3;
  subtype Month is Iso_Index
     range Year'Last + 2 .. Year'Last + 3;
  subtype Day is Iso_Index
     range Month'Last + 2 .. Month'Last + 3;
  Iso_Date : String (Iso_Index) := "2024-03-27";

begin
  Put_Line (Iso_Date (Year));  --  2024
  Put_Line (Iso_Date (Month)); --  03
  Put_Line (Iso_Date (Day));   --  27
```

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 TwoD_T is array (Index_T) of OneD_T;
A : TwoD_T;
B : OneD_T;
```

Which statement(s) is (are) legal?

A. `B(1) := A(1,2) or A(4,3);`
B. `B := A(2) and A(4);`
C. `A(1..2)(4) := A(5..6)(8);`
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 TwoD_T is array (Index_T) of OneD_T;
A : TwoD_T;
B : OneD_T;
```

Which statement(s) is (are) legal?

A. *`B(1) := A(1,2) or A(4,3);`*
B. *`B := A(2) and A(4);`*
C. `A(1..2)(4) := A(5..6)(8);`
D. *`B(3..4) := B(4..5)`*

Explanations

A. All objects are just Boolean values
B. An element of `A` is the same type as `B`
C. Slice must be of outermost array
D. Slicing allowed on single-dimension arrays

Looping Over Array Components

Note on Default Initialization for Array Types

- In Ada, objects are not initialized by default
- To initialize an array, you can initialize each element
  - But if the array type is used in multiple places, it would be better to initialize at the type level
  - 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

- 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

- 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

- 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;
```

Quiz

```
declare
   type Array_T is array (1..5) of Integer
      with Default_Component_Value => 1;
   A : Array_T;
begin
   for I in A'First + 1 .. A'Last - 1 loop
      A (I) := I * A'Length;
   end loop;
   for I of reverse A loop
      Put (I'Image);
   end loop;
end;
```

Which output is correct?

A. 1 10 15 20 1
B. 1 20 15 10 1
C. 0 10 15 20 0
D. 25 20 15 10 5

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

Quiz

```
declare
   type Array_T is array (1..5) of Integer
      with Default_Component_Value => 1;
   A : Array_T;
begin
   for I in A'First + 1 .. A'Last - 1 loop
      A (I) := I * A'Length;
   end loop;
   for I of reverse A loop
      Put (I'Image);
   end loop;
end;
```

Which output is correct?

A. 1 10 15 20 1
B. ***1 20 15 10 1***
C. 0 10 15 20 0
D. 25 20 15 10 5

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 arrays of composite component types

```
type Col_T is array (1 .. 3) of Float;
type Matrix_T is array (1 .. 3) of Col_T;
Matrix : Matrix_T := (1 =>  (1.2, 1.3, 1.4),
                      2 =>  (2.5, 2.6, 2.7),
                      3 =>  (3.8, 3.9, 3.0));
```

Defaults Within Array Aggregates

- 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)

Aggregates in Ada 2022

Ada 2022

- Ada 2022 allows us to use square brackets **"[...]"** in defining aggregates

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

  - So common aggregates can use either square brackets or parentheses

```
Ada2012 : Array_T := (1, 2, 3);
Ada2022 : Array_T := [1, 2, 3];
```

- But square brackets help in more problematic situations
  - Empty array

```
Ada2012 : Array_T := (1..0 => 0);
Illegal : Array_T := ();
Ada2022 : Array_T := [];
```

  - Single element array

```
Ada2012 : Array_T := (1 => 5);
Illegal : Array_T := (5);
Ada2022 : Array_T := [5];
```

Iterated Component Association

Ada 2022

- With Ada 2022, we can create aggregates with *iterators*
  - Basically, an inline looping mechanism
- Index-based iterator

```
type Array_T is array (positive range <>) of Integer;
Object1 : Array_T(1..5) := (for J in 1 .. 5 => J * 2);
Object2 : Array_T(1..5) := (for J in 2 .. 3 => J,
                            5 => -1,
                            others => 0);
```

  - `Object1` will get initialized to the squares of 1 to 5
  - `Object2` will give the equivalent of `(0, 2, 3, 0, -1)`
- Component-based iterator

```
Object2 := [for Item of Object => Item * 2];
```

  - `Object2` will have each element doubled

More Information on Iterators

Ada 2022

- You can nest iterators for arrays of arrays

```
type Col_T is array (1 .. 3) of Integer;
type Matrix_T is array (1 .. 3) of Col_T;
Matrix : Matrix_T :=
   [for J in 1 .. 3 =>
      [for K in 1 .. 3 => J * 10 + K]];
```

- You can even use multiple iterators for a single dimension array

```
Ada2012 : Array_T(1..5) :=
   [for I in 1 .. 2 => -1,
    for J in 4 ..5 => 1,
    others => 0];
```

- Restrictions
  - You cannot mix index-based iterators and component-based iterators in the same aggregate
  - You still cannot have overlaps or missing values

Delta Aggregates

Ada 2022

```
type Coordinate_T is array (1 .. 3) of Float;
Location : constant Coordinate_T := (1.0, 2.0, 3.0);
```

- Sometimes you want to copy an array with minor modifications
  - Prior to Ada 2022, it would require two steps

```
declare
   New_Location : Coordinate_T := Location;
begin
   New_Location(3) := 0.0;
   -- OR
   New_Location := (3 => 0.0, others => <>);
end;
```

- Ada 2022 introduces a *delta aggregate*
  - Aggregate indicates an object plus the values changed - the *delta*

```
New_Location : Coordinate_T := [Location with delta 3 => 0.0];
```

- Notes
  - You can use square brackets or parentheses
  - Only allowed for single dimension arrays

This works for records as well (see that chapter)

Detour - 'Image for Complex Types

'Image Attribute

Ada 2022

- Previously, we saw the string attribute `'Image` is provided for scalar types
  - e.g. `Integer'Image(10+2)` produces the string **" 12"**
- Starting with Ada 2022, the `Image` attribute can be used for any type

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Main is
   type Colors_T is (Red, Yellow, Green);
   type Array_T is array (Colors_T) of Boolean;
   Object : Array_T :=
     (Green  => False,
      Yellow => True,
      Red    => True);
begin
   Put_Line (Object'Image);
end Main;
```

Yields an output of

```
[TRUE, TRUE, FALSE]
```

Overriding the 'Image Attribute

Ada 2022

- We don't always want to rely on the compiler defining how we print a complex object
- We can define it - by using `'Image` and attaching a procedure to the `Put_Image` aspect

```
type Colors_T is (Red, Yellow, Green);
type Array_T is array (Colors_T) of Boolean with
  Put_Image => Array_T_Image;
```

Defining the 'Image Attribute

Ada 2022

- Then we need to declare the procedure

```
procedure Array_T_Image
  (Output : in out Ada.Strings.Text_Buffers.Root_Buffer_Type'Class;
   Value  :        Array_T);
```

  - Which uses the `Ada.Strings.Text_Buffers.Root_Buffer_Type` as an output buffer
  - (No need to go into detail here other than knowing you do `Output.Put` to add to the buffer)

- And then we define it

```
procedure Array_T_Image
  (Output : in out Ada.Strings.Text_Buffers.Root_Buffer_Type'Class;
   Value  :        Array_T) is
begin
   for Color in Value'Range loop
      Output.Put (Color'Image & "=>" & Value (Color)'Image & ASCII.LF);
   end loop;
end Array_T_Image;
```

Using the 'Image Attribute

Ada 2022

- Now, when we call Image we get our "pretty-print" version

```
with Ada.Text_IO; use Ada.Text_IO;
with Types; use Types;
procedure Main is
   Object : Array_T := (Green  => False,
                        Yellow => True,
                        Red    => True);
begin
   Put_Line (Object'Image);
end Main;
```

  - Generating the following output

```
RED=>TRUE
YELLOW=>TRUE
GREEN=>FALSE
```

- Note this redefinition can be used on any type, even the scalars that have always had the attribute

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

Arrays of Arrays

- 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;
   type Names_T is array (1 .. 3) of Name_T;
   Weekly_Staff : array (Days_Of_Week_T) of Names_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 loop
      Put_Line (Day'Image);
      for Staff of Weekly_Staff(Day) loop
         Put_Line ("  " & String (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
   Component1 : Integer;
   Component2 : 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(s) is (are) 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(s) is (are) 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. Run-time 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. Run-time 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. Run-time 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. Run-time error

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

Quiz

```
type Nested_T is record
   Component : 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) 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
   Component : 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) 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

Delta Aggregates

Ada 2022

- A Record can use a *delta aggregate* just like an array

```
type Coordinate_T is record
   X, Y, Z : Float;
end record;
Location : constant Coordinate_T := (1.0, 2.0, 3.0);
```

- Prior to Ada 2022, you would copy and then modify

```
declare
   New_Location : Coordinate_T := Location;
begin
   New_Location.Z := 0.0;
   -- OR
   New_Location := (Z => 0.0, others => <>);
end;
```

- Now in Ada 2022 we can just specify the change during the copy

```
New_Location : Coordinate_T := (Location with delta Z => 0.0);
```

Note for record delta aggregates you must use named notation

Default Values

Component Default Values

```ada
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

- 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

- 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

```
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

```
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`

Variant Records

Variant Record Types

- *Variant record* can use a **discriminant** to specify alternative lists of components
  - Also called *discriminated record* type
  - Different **objects** may have **different** components
  - All objects **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

Immutable Variant Record

- Discriminant must be set at creation time and cannot be modified

```
2  type Person_Group is (Student, Faculty);
3  type Person (Group : Person_Group) is
4  record
5     --  Components common across all discriminants
6     --  (must appear before variant part)
7     Age : Positive;
8     case Group is --  Variant part of record
9        when Student => -- 1st variant
10          Gpa  : Float range 0.0 .. 4.0;
11       when Faculty => -- 2nd variant
12          Pubs : Positive;
13    end case;
14 end record;
```

- In a variant record, a discriminant can be used to specify the *variant part* (line 8)
  - Similar to case statements (all values must be covered)
  - Components listed will only be visible if choice matches discriminant
  - Component names need to be unique (even across discriminants)
  - Variant part must be end of record (hence only one variant part allowed)
- Discriminant is treated as any other component
  - But is a constant in an immutable variant record

Note that discriminants can be used for other purposes than the variant part

Immutable Variant Record Example

- Each object of `Person` has three components, but it depends on `Group`

```
Pat : Person (Student);
Sam : Person := (Faculty, 33, 5);
```

  - `Pat` has `Group`, `Age`, and `Gpa`
  - `Sam` has `Group`, `Age`, and `Pubs`
  - Aggregate specifies all components, including the discriminant

- Compiler can detect some problems, but more often clashes are run-time errors

```
procedure Do_Something (Param : in out Person) is
begin
  Param.Age := Param.Age + 1;
  Param.Pubs := Param.Pubs + 1;
end Do_Something;
```

  - `Pat.Pubs := 3;` would generate a compiler warning because compiler knows `Pat` is a `Student`
    - `warning: Constraint_Error will be raised at run time`
  - `Do_Something (Pat);` generates a run-time error, because only at runtime is the discriminant for `Param` known
    - `raised CONSTRAINT_ERROR : discriminant check failed`

- `Pat := Sam;` would be a compiler warning because the constraints do not match

Mutable Variant Record

- Type will become *mutable* if its discriminant has a *default value* **and** we instantiate the object without specifying a discriminant

```
type Person_Group is (Student, Faculty);
type Person (Group : Person_Group := Student) is -- default value
record
   Age : Positive;
   case Group is
      when Student =>
         Gpa  : Float range 0.0 .. 4.0;
      when Faculty =>
         Pubs : Positive;
   end case;
end record;
```

- `Pat : Person;` is **mutable**
- `Sam : Person (Faculty);` is **not mutable**
  - Declaring an object with an **explicit** discriminant value (`Faculty`) makes it immutable

Mutable Variant Record Example

- Each object of `Person` has three components, but it depends on `Group`

```
Pat : Person := (Student, 19, 3.9);
Sam : Person (Faculty);
```

- You can only change the discriminant of `Pat`, but only via a whole record assignment, e.g:

```
if Pat.Group = Student then
  Pat := (Faculty, Pat.Age, 1);
else
  Pat := Sam;
end if;
Update (Pat);
```

- But you cannot change the discriminant of `Sam`
  - `Sam := Pat;` will give you a run-time error if `Pat.Group` is not `Faculty`
    - And the compiler will not warn about this!

Quiz

```
type Variant_T (Sign : Integer) is record
    case Sign is
    when Integer'First .. -1 =>
        I : Integer;
        B : Boolean;
    when others =>
        N : Natural;
    end case;
end record;

Variant_Object : Variant_T (1);
```

Which component(s) does `Variant_Object` contain?

A. `Variant_Object.I, Variant_Object.B`
B. `Variant_Object.N`
C. None: Compilation error
D. None: Run-time error

Quiz

```
type Variant_T (Sign : Integer) is record
    case Sign is
    when Integer'First .. -1 =>
        I : Integer;
        B : Boolean;
    when others =>
        N : Natural;
    end case;
end record;

Variant_Object : Variant_T (1);
```

Which component(s) does `Variant_Object` contain?

A. `Variant_Object.I, Variant_Object.B`
B. ***`Variant_Object.N`***
C. None: Compilation error
D. None: Run-time error

Quiz

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

Variant_Object : Variant_T (True);
```

Which component does `Variant_Object` contain?

A. `Variant_Object.F, Variant_Object.Flag`
B. `Variant_Object.F`
C. None: Compilation error
D. None: Run-time error

Quiz

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

Variant_Object : Variant_T (True);
```

Which component does `Variant_Object` contain?

A. `Variant_Object.F, Variant_Object.Flag`
B. `Variant_Object.F`
C. ***None: Compilation error***
D. None: Run-time error

The variant part cannot be followed by a component declaration (`Flag : Character` here)

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

```
10 Open (Source, "SomeFile.txt");
11 while not End_of_File (Source) loop
12   Get (Next_Char, From => Source);
13   if Found (Next_Char, Within => Buffer) then
14     Display (Next_Char);
15     Increment;
16   end if;
17 end loop;
```

  - Note that a subprogram without parameters (`Increment` on line 15) does not allow an empty set of parentheses

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);
```

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
- Note that the parameter mode `aliased` will force pass-by-reference
  - This mode is discussed in the **Access Types** module

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'Length /= Right'Length`
- 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
   pragma Assert (Left'Length = Right'Length);

   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;

   return Result;
end Subtract;
```

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(s) is (are) 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(s) is (are) 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

- 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

- 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

- 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

- 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
```

Main Subprograms

- Must be library subprograms
  - Not nested inside another subprogram
- No special subprogram unit name required
- Can be many per project
- Can always 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;
```

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

- 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

- 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.***

Explanation

A. They **can** be nested subprograms (just like any other subprogram)

B. As in other subprograms, the implementation can serve as the specification

C. Because they are expressions, the `return` statement is not allowed

D. An expression function does not allow assignment statements, but it can call another function that is **not** an expression function.

```
function Normal_Fun (Input  :     Character;
                     Output : out Integer)
                     return Boolean is
begin
   Output := Character'Pos (Input);
   return True;
end Normal_Fun;

function Expr_Fun (Input  :     Character;
                   Output : out Integer)
                   return Boolean is
   (Normal_Fun (Character'Succ (Input), Output));
```

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);
```

```
error:  writable actual for "Doubled" overlaps with actual for "Tripled"
```

Functions' Parameter Modes

- 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` in earlier versions of the language
  - 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 Example

Implementing a Simple "Set"

- We want to indicate which colors of the rainbow are in a **set**
  - If you remember from the *Basic Types* module, a type is made up of values and primitive operations
- Our values will be
  - Type indicating colors of the rainbow
  - Type to group colors
  - Mechanism to indicate which color is in our set
- Our primitive operations will be
  - Create a set
  - Add a color to the set
  - Remove a color from the set
  - Check if color is in set

Values for the Set

- Colors of the rainbow

```
type Color_T is (Red, Orange, Yellow, Green,
                 Blue, Indigo, Violet,
                 White, Black);
```

- Group of colors

```
type Group_Of_Colors_T is
   array (Positive range <>) of Color_T;
```

- Mechanism indicating which color is in the set

```
type Set_T is array (Color_T) of Boolean;
--  if array component at Color is True,
--  the color is in the set
```

Primitive Operations for the Set

- Create a set

```
function Make (Colors : Group_Of_Colors_T) return Set_T;
```

- Add a color to the set

```
procedure Add (Set     : in out Set_T;
               Color   :        Color_T);
```

- Remove a color from the set

```
procedure Remove (Set     : in out Set_T;
                  Color   :        Color_T);
```

- Check if color is in set

```
function Contains (Set   : Set_T;
                   Color : Color_T)
                   return Boolean;
```

Implementation of the Primitive Operations

- Implementation of the primitives is easy
  - We could do operations directly on Set_T, but that's not flexible

```
function Make (Colors : Group_Of_Colors_T) return Set_T is
   Set : Set_T := (others => False);
begin
   for Color of Colors loop
      Set (Color) := True;
   end loop;
   return Set;
end Make;

procedure Add (Set   : in out Set_T;
               Color :        Color_T) is
begin
   Set (Color) := True;
end Add;

procedure Remove (Set   : in out Set_T;
                  Color :        Color_T) is
begin
   Set (Color) := False;
end Remove;

function Contains (Set   : Set_T;
                   Color : Color_T)
                   return Boolean is
   (Set (Color));
```

Using our Set Construct

```
Rgb   : Set_T := Make ((Red, Green, Blue));
Light : Set_T := Make ((Red, Yellow, Green));

if Contains (Rgb, Black) then
   Remove (Rgb, Black);
else
   Add (Rgb, Black);
end if;
```

In addition, because of the operations available to arrays of Boolean, we can easily implement set operations

```
Union           : Set_T := Rgb or Light;
Intersection    : Set_T := Rgb and Light;
Difference      : Set_T := Rgb xor Light;
```

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!

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(s) is (are) 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(s) is (are) 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 its 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 to 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

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

Reminder: What is a Type?

- 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

```
package Types is
   type Integer_T is range -(2**63) .. 2**63-1 with Size => 64;
   procedure Increment_With_Truncation (Val : in out Integer_T);
   procedure Increment_With_Rounding (Val : in out Integer_T);
end Types;
```

Simple Derivation

Simple Type Derivation

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

```
type Natural_T is new Integer_T range 0 .. Integer_T'Last;
```

- `Natural_T` inherits from:
  - The data **representation** of the parent
    - Integer based, 64 bits
  - The **primitives** of the parent
    - `Increment_With_Truncation` and `Increment_With_Rounding`
- The types are not the same

```
I_Obj : Integer_T := 0;
N_Obj : Natural_T := 0;
```

  - `I_Obj := N_Obj;` → generates a compile error

    expected type "Integer_T" defined at line 2
  - But a child can be converted to the parent
    - `I_Obj := Integer_T (N_Obj);`

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 Positive_T is new Natural_T range 1 .. Natural_T'Last;
```

- Unconstrained types can be constrained

```
type Arr_T is array (Integer range <>) of Integer;
type Ten_Elem_Arr_T is new Arr_T (1 .. 10);
type Rec_T (Size : Integer) is record
   Elem : Arr_T (1 .. Size);
end record;
type Ten_Elem_Rec_T is new Rec_T (10);
```

Primitives

Primitive Operations

- Primitive Operations are those subprograms associated with a type

```
type Integer_T is range -(2**63) .. 2**63-1 with Size => 64;
procedure Increment_With_Truncation (Val : in out Integer_T);
procedure Increment_With_Rounding (Val : in out Integer_T);
```

- Most types have some primitive operations defined by the language
  - e.g. equality operators for most types, numeric operators for integers and floats
- A primitive operation on the parent can receive an object of a child type with no conversion

```
declare
   N_Obj : Natural_T := 1234;
begin
   Increment_With_Truncation (N_Obj);
end;
```

General Rule for Defining a Primitive

- Primitives are subprograms
- Subprogram `S` is a primitive of type `T` if and only if:
  - `S` is declared in the scope of `T`
  - `S` uses type `T`
    - As a parameter
    - As its return type (for a `function`)
  - `S` is above *freeze-point* (see next section)
- Standard practice
  - Primitives should 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;
```

Primitive of Multiple Types

A subprogram can be a primitive of several types

```
package P is
   type Distance_T is range 0 .. 9999;
   type Percentage_T is digits 2 range 0.0 .. 1.0;
   type Units_T is (Meters, Feet, Furlongs);

   procedure Convert (Value  : in out Distance_T;
                      Source :        Units_T;
                      Result :        Units_T;
   procedure Shrink (Value   : in out Distance_T;
                     Percent :        Percentage_T);

end P;
```

- Convert and Shrink are primitives for Distance_T
- Convert is also a primitive of Units_T
- Shrink is also a primitive of Percentage_T

Creating Primitives for Children

- Just because we can inherit a primitive from out parent doesn't mean we want to
- We can create a new primitive (with the same name as the parent) for the child
  - Very similar to overloaded subprograms
  - But added benefit of visibility to grandchildren
- We can also remove a primitive (see next slide)

```
type Integer_T is range -(2**63) .. 2**63-1;
procedure Increment_With_Truncation (Val : in out Integer_T);
procedure Increment_With_Rounding (Val : in out Integer_T);

type Child_T is new Integer_T range -1000 .. 1000;
procedure Increment_With_Truncation (Val : in out Child_T);

type Grandchild_T is new Child_T range -100 .. 100;
procedure Increment_With_Rounding (Val : in out Grandchild_T);
```

Overriding Indications

- **Optional** indications
- Checked by compiler

```
type Child_T is new Integer_T range -1000 .. 1000;
procedure Increment_With_Truncation
   (Val : in out Child_T);
procedure Just_For_Child
   (Val : in out Child_T);
```

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

```
overriding procedure Increment_With_Truncation
   (Val : in out Child_T);
```

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

```
not overriding procedure Just_For_Child
   (Val : in out Child_T);
```

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

```
overriding procedure Just_For_Child
   (Val : in out Grandchild_T) is abstract;
```

- Using `overriding` or `not overriding` incorrectly will generate a compile error

Quiz

```
type T is new Integer;
```

Which operator(s) definition(s) is (are) legal?

A. `function "+" (V : T) return Boolean is (V /= 0)`
B. `function "+" (A, B : T) return T is (A + B)`
C. `function "=" (A, B : T) return T is (A - B)`
D. `function ":=" (A : T) return T is (A)`

Quiz

```
type T is new Integer;
```

Which operator(s) definition(s) is (are) legal?

A. *`function "+" (V : T) return Boolean is (V /= 0)`*
B. `function "+" (A, B : T) return T is (A + B)`
C. *`function "=" (A, B : T) return T is (A - B)`*
D. `function ":=" (A : T) return T is (A)`

B. Infinite recursion
C. Unlike some languages, there is no assignment operator

Freeze Point

What is the "Freeze Point"?

- Ada doesn't explicitly identify the end of the "scope" of a type
  - The compiler needs to know it for determining primitive operations
  - Also needed for other situations (described elsewhere)
- This end is the implicit **freeze point** occurring whenever:
  - A **variable** of the type is **declared**
  - The type is **derived**
  - The **end of the scope** is reached
- Subprograms past this "freeze point" are not primitive operations

```
type Parent is Integer;
procedure Prim (V : Parent);

type Child is new Parent;

-- Parent has been derived, so it is frozen.
-- Prim2 is not a primitive
procedure Prim2 (V : Parent);

V : Child;

-- Child used in an object declaration, so it is frozen
-- Prim3 is not a primitive
procedure Prim3 (V : Child);
```

Debugging Type Freeze

- Freeze → Type **completely** defined
- Compiler does **need** to determine the freeze point
  - To instantiate, derive, get info on the type (`'Size`)...
  - Freeze rules are a guide to place it
  - Actual choice is more technical
    - May contradict the standard
- `-gnatDG` to get **expanded** source
  - **Pseudo-Ada** debug information

pkg.ads

```
type Up_To_Eleven is range 0 .. 11;
```

<obj>/pkg.ads.dg

```
type example__up_to_eleven_t is range 0 .. 11;              -- type declaration
[type example__Tup_to_eleven_tB is new short_short_integer] -- representation
freeze example__Tup_to_eleven_tB []                         -- freeze representation
freeze example__up_to_eleven_t []                           -- freeze representation
```

Quiz

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

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

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

type Grandchild is new Child range 3 .. 98;

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

Which are `Parent`'s primitives?

A. `Proc_A`
B. `Proc_B`
C. `Proc_C`
D. No primitives of `Parent`

Quiz

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

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

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

type Grandchild is new Child range 3 .. 98;

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

Which are `Parent`'s primitives?

A. ***`Proc_A`***
B. `Proc_B`
C. `Proc_C`
D. No primitives of `Parent`

Explanations

A. Correct
B. Freeze: `Parent` has been derived
C. Freeze: scope change
D. Incorrect

Summary

Summary

- *Primitive* of a type
  - Subprogram above **freeze-point** that takes or returns 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

Exceptions

Introduction

Rationale for Exceptions

- Textual separation from normal processing
- Rigorous Error Management
  - Cannot be ignored, unlike status codes from routines
  - Example: running out of gasoline in an automobile

```
package Automotive is
  type Vehicle is record
    Fuel_Quantity, Fuel_Minimum : Float;
    Oil_Temperature : Float;
    ...
  end record;
  Fuel_Exhausted : exception;
  procedure Consume_Fuel (Car : in out Vehicle);
  ...
end Automotive;
```

Semantics Overview

- Exceptions become active by being *raised*
  - Failure of implicit language-defined checks
  - Explicitly by application
- Exceptions occur at run-time
  - A program has no effect until executed
- May be several occurrences active at same time
  - One per task
- Normal execution abandoned when they occur
  - Error processing takes over in response
  - Response specified by *exception handlers*
  - *Handling the exception* means taking action in response
  - Other tasks need not be affected

Semantics Example: Raising

```
package body Automotive is
  function Current_Consumption return Float is
    ...
  end Current_Consumption;
  procedure Consume_Fuel (Car : in out Vehicle) is
  begin
    if Car.Fuel_Quantity <= Car.Fuel_Minimum then
      raise Fuel_Exhausted;
    else -- decrement quantity
      Car.Fuel_Quantity := Car.Fuel_Quantity -
                           Current_Consumption;
    end if;
  end Consume_Fuel;
  ...
end Automotive;
```

Semantics Example: Handling

```
procedure Joy_Ride is
  Hot_Rod : Automotive.Vehicle;
  Bored : Boolean := False;
  use Automotive;
begin
  while not Bored loop
    Steer_Aimlessly (Bored);
    -- error situation cannot be ignored
    Consume_Fuel (Hot_Rod);
  end loop;
  Drive_Home;
exception
  when Fuel_Exhausted =>
    Push_Home;
end Joy_Ride;
```

Handler Part Is Skipped Automatically

- If no exceptions are active, returns normally

```
begin
   ...
-- if we get here, skip to end
exception
  when Name1 =>
   ...
  when Name2 | Name3 =>
   ...
  when Name4 =>
   ...
end;
```

Handlers

Exception Handler Part

- Contains the exception handlers within a frame
    - Within block statements, subprograms, tasks, etc.
- Separates normal processing code from abnormal
- Starts with the reserved word `exception`
- Optional

```
begin
  sequence_of_statements
[ exception
    exception_handler
    { exception handler } ]
end
```

Exception Handlers Syntax

- Associates exception names with statements to execute in response
- If used, **others** must appear at the end, by itself
  - Associates statements with all other exceptions
- Syntax

```
exception_handler ::=
  when exception_choice { | exception_choice } =>
    sequence_of_statements
exception_choice ::= exception_name | others
```

Similarity to Case Statements

- Both structure and meaning
- Exception handler

```
...
exception
  when Constraint_Error | Storage_Error | Program_Error =>
  ...
  when others =>
  ...
end;
```

- Case statement

```
case exception_name is
  when Constraint_Error | Storage_Error | Program_Error =>
  ...
  when others =>
  ...
end case;
```

Handlers Don't "Fall Through"

```
begin
   ...
   raise Name3;
   -- code here is not executed
   ...
exception
   when Name1 =>
      -- not executed
      ...
   when Name2 | Name3 =>
      -- executed
      ...
   when Name4 =>
      -- not executed
      ...
end;
```

When an Exception Is Raised

- Normal processing is abandoned
- Handler for active exception is executed, if any
- Control then goes to the caller
- If handled, caller continues normally, otherwise repeats the above

- Caller

```
...
Joy_Ride;
Do_Something_At_Home;
...
```

- Callee

```
procedure Joy_Ride is
   ...
begin
   ...
   Drive_Home;
exception
   when Fuel_Exhausted =>
      Push_Home;
end Joy_Ride;
```

Handling Specific Statements' Exceptions

```
begin
  loop
    Prompting : loop
      Put (Prompt);
      Get_Line (Filename, Last);
      exit when Last > Filename'First - 1;
    end loop Prompting;
    begin
      Open (F, In_File, Filename (1..Last));
      exit;
    exception
      when Name_Error =>
        Put_Line ("File '" & Filename (1..Last) &
                  "' was not found.");
    end;
  end loop;
```

Exception Handler Content

- No restrictions
  - Block statements, subprogram calls, etc.
- Do whatever makes sense

```
begin
  ...
exception
  when Some_Error =>
    declare
      New_Data : Some_Type;
    begin
      P (New_Data);
      ...
    end;
end;
```

Quiz

```
procedure Main is
   A, B, C, D : Integer range 0 .. 100;
begin
   A := 1; B := 2; C := 3; D := 4;
   begin
      D := A - C + B;
   exception
      when others => Put_Line ("One");
                     D := 1;
   end;
   D := D + 1;
   begin
      D := D / (A - C + B);
   exception
      when others => Put_Line ("Two");
                     D := -1;
   end;
exception
   when others =>
      Put_Line ("Three");
end Main;
```

What will get printed?

A. One, Two, Three
B. Two, Three
C. Two
D. Three

Quiz

```
1  procedure Main is
2     A, B, C, D : Integer range 0 .. 100;
3  begin
4     A := 1; B := 2; C := 3; D := 4;
5     begin
6        D := A - C + B;
7     exception
8        when others => Put_Line ("One");
9                       D := 1;
10    end;
11    D := D + 1;
12    begin
13       D := D / (A - C + B);
14    exception
15       when others => Put_Line ("Two");
16                      D := -1;
17    end;
18 exception
19    when others =>
20       Put_Line ("Three");
21 end Main;
```

What will get printed?

A. One, Two, Three
B. ***Two, Three***
C. Two
D. Three

Explanations

A. Although `(A - C)` is not in the range of `natural`, the range is only checked on assignment, which is after the addition of `B`, so `One` is never printed
B. Correct
C. If we reach `Two`, the assignment on line 16 will cause `Three` to be reached
D. Divide by 0 on line 13 causes an exception, so `Two` must be called

Implicitly and Explicitly Raised Exceptions

Implicitly-Raised Exceptions

- Correspond to language-defined checks
- Can happen by statement execution

```
K := -10;  -- where K must be greater than zero
```

- Can happen by declaration elaboration

```
Doomed : array (Positive) of Big_Type;
```

Some Language-Defined Exceptions

- `Constraint_Error`
  - Violations of constraints on range, index, etc.
- `Program_Error`
  - Runtime control structure violated (function with no return ...)
- `Storage_Error`
  - Insufficient storage is available
- For a complete list see RM Q-4

Explicitly-Raised Exceptions

- Raised by application via **raise** statements
  - Named exception becomes active
- Syntax

```
raise_statement ::= raise; |
   raise exception_name
   [with string_expression];
```

  Note "with string_expression" only available in Ada 2005 and later
- A **raise** by itself is only allowed in handlers

```
if Unknown (User_ID) then
  raise Invalid_User;
end if;

if Unknown (User_ID) then
  raise Invalid_User
     with "Attempt by " &
          Image (User_ID);
end if;
```

User-Defined Exceptions

User-Defined Exceptions

- Syntax

```
defining_identifier_list : exception;
```

- Behave like predefined exceptions
  - Scope and visibility rules apply
  - Referencing as usual
  - Some minor differences
- Exception identifiers' use is restricted
  - `raise` statements
  - Handlers
  - Renaming declarations

User-Defined Exceptions Example

- An important part of the abstraction
- Designer specifies how component can be used

```
package Stack is
  Underflow, Overflow : exception;
  procedure Push (Item : in Integer);
  ...
end Stack;

package body Stack is
  procedure Push (Item : in Integer) is
  begin
    if Top = Index'Last then
      raise Overflow;
    end if;
    Top := Top + 1;
    Values (Top) := Item;
  end Push;
...
```

Propagation

Propagation

- Control does not return to point of raising
  - Termination Model
- When a handler is not found in a block statement
  - Re-raised immediately after the block
- When a handler is not found in a subprogram
  - Propagated to caller at the point of call
- Propagation is dynamic, back up the call chain
  - Not based on textual layout or order of declarations
- Propagation stops at the main subprogram
  - Main completes abnormally unless handled

Propagation Demo

```
procedure Do_Something is
  Error : exception;
  procedure Unhandled is
  begin
    Maybe_Raise (1);
  end Unhandled;
  procedure Handled is
  begin
    Unhandled;
    Maybe_Raise (2);
  exception
    when Error =>
      Print ("Handle 1 or 2");
  end Handled;
begin -- Do_Something
  Maybe_Raise (3);
  Handled;
exception
  when Error =>
    Print ("Handle 3");
end Do_Something;
```

Termination Model

- When control goes to handler, it continues from here

```
procedure Joy_Ride is
begin
   loop
       Steer_Aimlessly;

       -- If next line raises Fuel_Exhausted, go to handler
       Consume_Fuel;
   end loop;
exception
  when Fuel_Exhausted => -- Handler
    Push_Home;
    -- Resume from here: loop has been exited
end Joy_Ride;
```

Quiz

```
2  Main_Problem : exception;
3  I : Integer;
4  function F (P : Integer) return Integer is
5  begin
6    if P > 0 then
7      return P + 1;
8    elsif P = 0 then
9      raise Main_Problem;
10   end if;
11 end F;
12 begin
13   I := F(Input_Value);
14   Put_Line ("Success");
15 exception
16   when Constraint_Error => Put_Line ("Constraint Error");
17   when Program_Error    => Put_Line ("Program Error");
18   when others           => Put_Line ("Unknown problem");
```

What will get printed if `Input_Value` on line 13 is `Integer'Last`?

A. `Unknown Problem`
B. `Success`
C. `Constraint Error`
D. `Program Error`

Quiz

```
2  Main_Problem : exception;
3  I : Integer;
4  function F (P : Integer) return Integer is
5  begin
6    if P > 0 then
7      return P + 1;
8    elsif P = 0 then
9      raise Main_Problem;
10   end if;
11 end F;
12 begin
13   I := F(Input_Value);
14   Put_Line ("Success");
15 exception
16   when Constraint_Error => Put_Line ("Constraint Error");
17   when Program_Error    => Put_Line ("Program Error");
18   when others           => Put_Line ("Unknown problem");
```

What will get printed if `Input_Value` on line 13 is `Integer'Last`?

A. `Unknown Problem`
B. `Success`
C. *`Constraint Error`*
D. `Program Error`

Explanations

A. `"Unknown Problem"` is printed by the **`when others`** due to the raise on line 9 when `P` is 0
B. `"Success"` is printed when $0 <$ `P` $<$ `Integer'Last`
C. Trying to add 1 to `P` on line 7 generates a `Constraint_Error`
D. `Program_Error` will be raised by `F` if `P` < 0 (no **`return`** statement found)

Exceptions As Objects

Exceptions Are Not Objects

- May not be manipulated
  - May not be components of composite types
  - May not be passed as parameters
- Some differences for scope and visibility
  - May be propagated out of scope

But You Can Treat Them As Objects

- For raising and handling, and more
- Standard Library

```
package Ada.Exceptions is
  type Exception_Id is private;
  procedure Raise_Exception (E : Exception_Id;
                             Message : String := "");
  ...
  type Exception_Occurrence is limited private;
  function Exception_Name (X : Exception_Occurrence)
      return String;
  function Exception_Message (X : Exception_Occurrence)
      return String;
  function Exception_Information (X : Exception_Occurrence)
      return String;
  procedure Reraise_Occurrence (X : Exception_Occurrence);
  procedure Save_Occurrence (
    Target : out Exception_Occurrence;
    Source : Exception_Occurrence);
  ...
end Ada.Exceptions;
```

Exception Occurrence

- Syntax associates an object with active exception

```
when defining_identifier : exception_name ... =>
```

- A constant view representing active exception
- Used with operations defined for the type

```
exception
  when Caught_Exception : others =>
    Put (Exception_Name (Caught_Exception));
```

Exception_Occurrence Query Functions

- **Exception_Name**
  - Returns full expanded name of the exception in string form
    - Simple short name if space-constrained
  - Predefined exceptions appear as just simple short name
- **Exception_Message**
  - Returns string value specified when raised, if any
- **Exception_Information**
  - Returns implementation-defined string content
  - Should include both exception name and message content
  - Presumably includes debugging information
    - Location where exception occurred
    - Language-defined check that failed (if such)

Exception ID

- For an exception identifier, the *identity* of the exception is `<name>'Identity`

```
Mine : exception
use Ada.Exceptions;
...
exception
   when Occurrence : others =>
      if Exception_Identity (Occurrence) = Mine'Identity
      then
         ...
```

Raise Expressions

Raise Expressions

- **Expression** raising specified exception **at run-time**

```
Foo : constant Integer := (case X is
                            when 1 => 10,
                            when 2 => 20,
                            when others => raise Error);
```

Lab

Exceptions Lab

(Simplified) Input Verifier

- Overview
  - Create an application that converts strings to numeric values
- Requirements
  - Create a package to define your numeric type
  - Define a primitive to convert a string to your numeric type
    - The primitive should raise your own exceptions; one for out-of-range and one for illegal string
  - Main program should run multiple tests on the primitive

Exceptions Lab Solution - Numeric Types

```ada
package Numeric_Types is
   Illegal_String : exception;
   Out_Of_Range   : exception;

   Max_Int : constant := 2**15;
   type Integer_T is range -(Max_Int) .. Max_Int - 1;

   function Value (Str : String) return Integer_T;
end Numeric_Types;

package body Numeric_Types is

   function Legal (C : Character) return Boolean is
   begin
      return
        C in '0' .. '9' or C = '+' or C = '-' or C = '_' or C = 'e' or C = 'E';
   end Legal;

   function Value (Str : String) return Integer_T is
   begin
      for I in Str'Range loop
         if not Legal (Str (I)) then
            raise Illegal_String;
         end if;
      end loop;
      return Integer_T'Value (Str);
   exception
      when Constraint_Error =>
         raise Out_Of_Range;
   end Value;

end Numeric_Types;
```

Exceptions Lab Solution - Main

```ada
with Ada.Text_IO;
with Numeric_Types;
procedure Main is

   procedure Print_Value (Str : String) is
      Value : Numeric_Types.Integer_T;
   begin
      Ada.Text_IO.Put (Str & " => ");
      Value := Numeric_Types.Value (Str);
      Ada.Text_IO.Put_Line (Numeric_Types.Integer_T'Image (Value));
   exception
      when Numeric_Types.Out_Of_Range =>
         Ada.Text_IO.Put_Line ("Out of range");
      when Numeric_Types.Illegal_String =>
         Ada.Text_IO.Put_Line ("Illegal entry");
   end Print_Value;

begin
   Print_Value ("123");
   Print_Value ("2_3_4");
   Print_Value ("-345");
   Print_Value ("+456");
   Print_Value ("1234567890");
   Print_Value ("123abc");
   Print_Value ("12e3");
end Main;
```

Summary

Exceptions Are Not Always Appropriate

- What does it mean to have an unexpected error in a safety-critical application?
    - Maybe there's no reasonable response

Relying on Exception Raising Is Risky

- They may be **suppressed**
  - By runtime environment
  - By build switches
- Not recommended

```
function Tomorrow (Today : Days) return Days is
begin
  return Days'Succ (Today);
exception
  when Constraint_Error =>
    return Days'First;
end Tomorrow;
```

- Recommended

```
function Tomorrow (Today : Days) return Days is
begin
  if Today = Days'Last then
    return Days'First;
  else
    return Days'Succ (Today);
  end if;
end Tomorrow;
```

Summary

- Should be for unexpected errors
- Give clients the ability to avoid them
- If handled, caller should see normal effect
  - Mode `out` parameters assigned
  - Function return values provided
- Package **Ada.Exceptions** provides views as objects
  - For both raising and special handling
  - Especially useful for debugging
- Checks may be suppressed

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

```
-- with aspects
type Some_Integer_T is range 1 .. 10
   with Object_Size => 8,
        Value_Size  => 4,
        Alignment   => 1;

-- with representation clauses
type Another_Integer_T is range 1 .. 10;
for Another_Integer_T'Object_Size use 8;
for Another_Integer_T'Value_Size  use 4;
for Another_Integer_T'Alignment   use 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

```
-- with aspect
type Small_T is range 1 .. 4
   with Size => 3;

-- with representation clause
type Another_Small_T is range 1 .. 4;
for Another_Small_T'Size use 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`

```
-- with aspects
type Some_T is range 1 .. 4
   with Value_Size  => 3,
        Object_Size => 8;

-- with representation clauses
type Another_T is range 1 .. 4;
for Another_T'Value_Size use 3;
for Another_T'Object_Size use 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

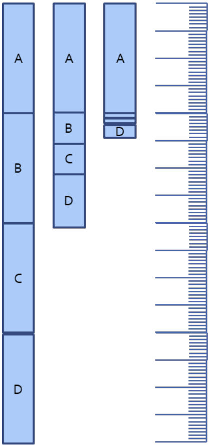
```
-- with aspects
type Aligned_T is range 1 .. 4
   with Size      => 4,
        Alignment => 8;

-- with representation clauses
type Another_Aligned_T is range 1 .. 4;
for Another_Aligned_T'Size use 4;
for Another_Aligned_T'Alignment use 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
```

Enum Representation Clauses

- Can specify representation for each value
- Representation must have increasing number

```
type E is (A, B, C);
for E use (A => 2, B => 4, C => 8);
```

- Can use `E'Enum_Rep (A) = 2`
- Can use `E'Enum_Val (2) = A`

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...
- Components 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;
```

Unchecked Unions

- Allows replicating C's **union** with **discriminated** records
- Discriminant is **not stored**
- No discriminant check
- Object must be **mutable**

```
type R (Is_Float : Boolean := False) is record
    case Is_Float is
    when True =>
        F : Float;
    when False =>
        I : Integer;
    end case;
end record
    with Unchecked_Union;

O : R := (Is_Float => False, I => 1);
F : Float := R.F; --  no check!
```

Array Representation Clauses

- Component_Size for array's **component's** size

```
-- with aspect
type Array_T is array (1 .. 1000) of Boolean
    with Component_Size => 2;

-- with representation clause
type Another_Array_T is array (1 .. 1000) of Boolean;
for Another_Array_T'Component_Size use 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

```
-- with aspect
type Array_T is array (1 .. 1000) of Boolean with
  Scalar_Storage_Order => System.Low_Order_First;

-- with representation clauses
type Record_T is record
   A : Integer;
   B : Boolean;
end record;
for Record_T use record
   A at 0 range 0 .. 31;
   B at 0 range 32 .. 33;
end record;
for Record_T'Bit_Order use System.High_Order_First;
for Record_T'Scalar_Storage_Order use System.High_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
     Component1 : Unsigned_8;
     Component2 : Unsigned_16;
     Component3 : Unsigned_8;
end record;
type Packed_Rec_T is new Rec_T;
for Packed_Rec_T use record
   Component1 at 0 range  0 ..  7;
   Component2 at 0 range  8 .. 23;
   Component3 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

```
Use_Aspect     : Unsigned_32 with
   Address => 16#1234_ABCD#;

Use_Rep_Clause : Unsigned_32;
for Use_Rep_Clause'Address use 16#5678_1234#;
```

- Very useful to declare I/O registers
  - For that purpose, the object should be declared volatile:

```
Use_Aspect     : Unsigned_32 with
   Volatile,
   Address => 16#1234_ABCD#;

Use_Rep_And_Pragma : Unsigned_32;
for Use_Rep_And_Pragma'Address use 16#5678_1234#;
pragma Volatile (Use_Rep_And_Pragma);
```

- Useful to read a value anywhere

```
function Get_Byte (Addr : Address) return Unsigned_8 is
  V : Unsigned_8 with Address => Addr, Volatile;
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

```
V : Unsigned_32 with
    Address =>
        System.Storage_Elements.To_Address (16#120#);
```

- GNAT specific attribute 'To_Address
  - Handy but not portable

```
V : Unsigned_32 with
    Address => System'To_Address (16#120#);
```

Volatile

- The `Volatile` property can be set using an aspect or a pragma
- Ada also allows volatile types as well as objects

```
type Volatile_U32 is mod 2**32 with Volatile;
type Volatile_U16 is mod 2**16;
pragma Volatile (Volatile_U16);
```

- 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 Bit_Array_T is array (Integer range <>) of Boolean
   with Component_Size => 1;

-- objects can be referenced elsewhere
Object  : aliased Integer with Volatile;
Object2 : aliased Integer with Volatile;

Object_A : System.Address := Object'Address;
Object_I : Integer_Address := To_Integer (Object_A);

--  This overlays Bit_Array_Object onto Object in memory
Bit_Array_Object : aliased Bit_Array_T (1 .. Object'Size)
   with Address => Object_A;

Object2_Alias : aliased Integer
   --  Trust me, I know what I'm doing, this is Object2
   with Address => To_Address (Object_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

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 component
    - Two other components
    - 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)

```ada
   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

Supplementary Resource: Inline ASM

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;
```

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

Basic Syntax and Nomenclature

- Spec
  - Basic declarative items **only**
  - e.g. no subprogram bodies

  ```
  package name is
     {basic_declarative_item}
  end [name];
  ```

- Body

```
package body name is
   declarative_part
end [name];
```

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!

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 Some_Package is
  -- exported declarations of
  --   types, variables, subprograms ...
end Some_Package;
```

- Items in the body are never externally visible
  - Compiler prevents external references

```
package body Some_Package is
  -- hidden declarations of
  --   types, variables, subprograms ...
  -- implementations of exported subprograms etc.
end Some_Package;
```

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 Other Packages

with Clause

- When package `Client` needs access to package `Server`, it uses a **with** clause
  - Specify the library units that `Client` depends upon
  - The "context" in which the unit is compiled
  - `Client`'s code gets **visibility** over `Server`'s specification
- Syntax (simplified)

```
context_clause ::= { context_item }
context_item ::= with_clause | use_clause
with_clause ::= with library_unit_name
                { , library_unit_name };
```

```
with Server; -- dependency
procedure Client is
```

Referencing Exported Items

- Achieved via "dot notation"
- Package Specification

```
package Float_Stack is
  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);
   ...
```

with Clause Syntax

- A library unit is a package or subprogram that is not nested within another unit
  - Typically in its own file(s)
    - e.g. for package `Test`, GNAT defaults to expect the spec in `test.ads` and body in `test.adb`)
- Only library units may appear in a `with` statement
  - Can be a package or a standalone subprogram
- Due to the `with` syntax, library units cannot be overloaded
  - If overloading allowed, which **P** would `with P;` refer to?

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
    Component : A.Something;
  end record;
end B;

with B; -- no "with" of A
procedure Foo is
  X : B.Something;
begin
  X.Component := ...
```

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 (V : 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 (V : out Integer) is null;
end P;
```

C.
```
package body P is
  Object_One : Integer;
  procedure One (V : out Integer) is
  begin
    V := Object_One;
  end One;
end P;
```

D.
```
package body P is
  procedure One (V : out Integer) is
  begin
    V := Object_One;
  end One;
end P;
```

Quiz

```
package P is
   Object_One : Integer;
   procedure One (V : 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 (V : out Integer) is null;`
   `end P;`

C.
```
package body P is
  Object_One : Integer;
  procedure One (V : out Integer) is
  begin
    V := Object_One;
  end One;
end P;
```

D. *`package body P is`*
   `procedure One (V : out Integer) is`
   `begin`
   `V := Object_One;`
   `end One;`
   `end P;`

A. Procedure One must have a body
B. Parameter V 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;
```

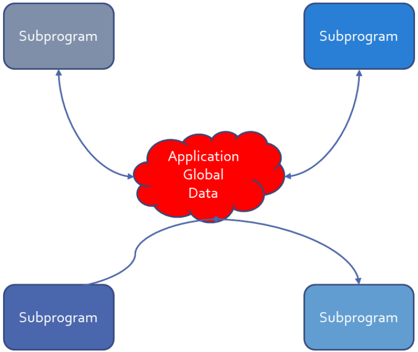
Group of Related Program Units

- 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

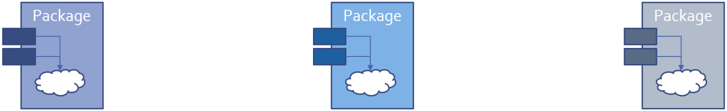


Packages and "Lifetime"

- Like a subprogram, objects declared directly in a package exist while the package is "in scope"
    - Whether the object is in the package spec or body
- Packages defined at the library level (not inside a subprogram) are always "in scope"
    - Including packages nested inside a package
- So package objects are considered "global data"
    - Putting variables in the spec exposes them to clients
        - Usually - in another module we talk about data hiding in the spec
    - Variables in the body can only be accessed from within the package body

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

```
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

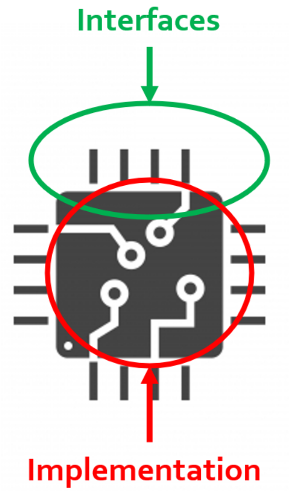
Introduction

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 Bounded_Stacks is
  type Stack is private;
  procedure Push (Item : in Integer; Onto : in out Stack);
  ...
private
  ...
  type Stack is record
     Top : Positive;
     ...
end Bounded_Stacks;
```

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 : Bounded_Stacks.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(s) is (are) legal?

A. `Component_A : Integer := Private_T'Pos (Private_T'First);`

B. `Component_B : Private_T := null;`

C. `Component_C : Private_T := 0;`

D. `Component_D : Integer := Private_T'Size;`

```
    end record;
```

Quiz

```
package P is
   type Private_T is private;

   type Record_T is record
```

Which component(s) is (are) legal?

A. `Component_A : Integer := Private_T'Pos (Private_T'First);`

B. `Component_B : Private_T := null;`

C. `Component_C : Private_T := 0;`

D. *`Component_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 and Recompilation

- Users can compile their code before the package body is compiled or even written
- 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(s) is (are) 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(s) is (are) 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

- Reminder: view is the *interface* you have on the type

- **User** of package has **Partial** view
  - Operations **exported** by package

- **Designer** of package has **Full** view
  - **Once** completion is reached
  - All operations based upon **full definition** of type

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.Instance := 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; -- Illegal since Vector is private
```

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)

```
function Find (Map : Map_T;
               Key : Key_T)
               return Integer is
begin
   for I in 1 .. Map.Length loop
      if Map.Values (I).Key = Key then
         return I;
      end if;
   end loop;
   return -1;
end Find;

procedure Add (Map         : in out Map_T;
               Key         :        Key_T;
               Description :        Colors.Color_Set_T;
               Success     :    out Boolean) is
   Index : constant Integer := Find (Map, Key);
begin
   Success := False;
   if Index not in Map.Values'Range then
      declare
         New_Item : constant Map_Element_T :=
           (Key         => Key,
            Description => Description);
      begin
         Map.Length              := Map.Length + 1;
         Map.Values (Map.Length) := New_Item;
         Success                 := True;
      end;
   end if;
end Add;

procedure Remove (Map     : in out Map_T;
                  Key     :        Key_T;
                  Success :    out Boolean) is
   Index : constant Integer := Find (Map, Key);
begin
   Success := False;
   if Index in Map.Values'Range then
      Map.Values (Index .. Map.Length - 1) :=
        Map.Values (Index + 1 .. Map.Length);
      Success                             := True;
   end if;
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
   Index : constant Integer := Find (Map, Key);
begin
   Success := False;
   if Index in Map.Values'Range then
      Map.Values (Index).Description := Description;
      Success                        := True;
   end if;
end Modify;

function Exists (Map : Map_T;
                 Key : Key_T)
                 return Boolean is
   (Find (Map, Key) in Map.Values'Range);

function Get (Map : Map_T;
              Key : Key_T)
              return Map_Element_T is
   Index   : constant Integer := Find (Map, Key);
   Ret_Val : Map_Element_T;
begin
   if Index in Map.Values'Range then
      Ret_Val := Map.Values (Index);
   end if;
   return Ret_Val;
end Get;

function Image (Item : Map_Element_T) return String is
  (Item.Key'Image & " => " & 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 I in 1 .. Flag.Length loop
      declare
         Item : constant Map_Element_T := Flag.Values (I);
         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

Limited Types

Introduction

Views

- Specify how values and objects may be manipulated
- Are implicit in much of the language semantics
  - Constants are just variables without any assignment view
  - Task types, protected types implicitly disallow assignment
  - Mode **in** formal parameters disallow assignment

```
Variable : Integer := 0;
...
-- P's view of X prevents modification
procedure P(X :  in  Integer) is
begin
    ...
end P;
...
P(Variable);
```

Limited Type Views' Semantics

- Prevents copying via predefined assignment
  - Disallows assignment between objects
  - Must make your own **copy** procedure if needed

```
type File is limited ...
...
F1, F2 : File;
...
F1 := F2; -- compile error
```

- Prevents incorrect comparison semantics
  - Disallows predefined equality operator
  - Make your own equality function = if needed

Inappropriate Copying Example

```
type File is ...
F1, F2 : File;
...
Open (F1);
Write (F1, "Hello");
-- What is this assignment really trying to do?
F2 := F1;
```

Intended Effects of Copying

```
type File is ...
F1, F2 : File;
...
Open (F1);
Write (F1, "Hello");
Copy (Source => F1, Target => F2);
```

Declarations

Limited Type Declarations

- Syntax
  - Additional keyword limited added to record type declaration

```
type defining_identifier is limited record
    component_list
end record;
```

- Are always record types unless also private
  - More in a moment...

Approximate Analog in C++

```
class Stack {
public:
  Stack ();
  void Push (int X);
  void Pop (int& X);
  ...
private:
  ...
  // assignment operator hidden
  Stack& operator= (const Stack& other);
}; // Stack
```

Spin Lock Example

```
with Interfaces;
package Multiprocessor_Mutex is
  -- prevent copying of a lock
  type Spin_Lock is limited record
    Flag : Interfaces.Unsigned_8;
  end record;
  procedure Lock  (This : in out Spin_Lock);
  procedure Unlock  (This : in out Spin_Lock);
  pragma Inline (Lock, Unlock);
end Multiprocessor_Mutex;
```

Parameter Passing Mechanism

- Always "by-reference" if explicitly limited
  - Necessary for various reasons (**task** and **protected** types, etc)
  - Advantageous when required for proper behavior
- By definition, these subprograms would be called concurrently
  - Cannot operate on copies of parameters!

```
procedure Lock   (This : in out Spin_Lock);
procedure Unlock (This : in out Spin_Lock);
```

Composites with Limited Types

- Composite containing a limited type becomes limited as well
  - Example: Array of limited elements
    - Array becomes a limited type
  - Prevents assignment and equality loop-holes

```
declare
  -- if we can't copy component S, we can't copy User_Type
  type User_Type is record -- limited because S is limited
    S : File;
    ...
  end record;
  A, B : User_Type;
begin
  A := B;  -- not legal since limited
  ...
end;
```

Quiz

```
type T is limited record
   I : Integer;
end record;

L1, L2 : T;
B : Boolean;
```

Which statement(s) is (are) legal?

A. `L1.I := 1`
B. `L1 := L2`
C. `B := (L1 = L2)`
D. `B := (L1.I = L2.I)`

Quiz

```
type T is limited record
   I : Integer;
end record;

L1, L2 : T;
B : Boolean;
```

Which statement(s) is (are) legal?

A. *`L1.I := 1`*
B. `L1 := L2`
C. `B := (L1 = L2)`
D. *`B := (L1.I = L2.I)`*

Quiz

```
type T is limited record
   I : Integer;
end record;
```

Which of the following declaration(s) is (are) legal?

A. `function "+" (A : T) return T is (A)`
B. `function "-" (A : T) return T is (I => -A.I)`
C. `function "=" (A, B : T) return Boolean is (True)`
D. `function "=" (A, B : T) return Boolean is (A.I = T'(I => B.I).I)`

Quiz

```
type T is limited record
   I : Integer;
end record;
```

Which of the following declaration(s) is (are) legal?

A. `function "+" (A : T) return T is (A)`
B. *`function "-" (A : T) return T is (I => -A.I)`*
C. *`function "=" (A, B : T) return Boolean is (True)`*
D. *`function "=" (A, B : T) return Boolean is (A.I = T'(I => B.I).I)`*

Quiz

```
package P is
   type T is limited null record;
   type R is record
      F1 : Integer;
      F2 : T;
   end record;
end P;

with P;
procedure Main is
   T1, T2 : P.T;
   R1, R2 : P.R;
begin
```

Which assignment(s) is (are) legal?

A. `T1    := T2;`
B. `R1    := R2;`
C. `R1.F1 := R2.F1;`
D. `R2.F2 := R2.F2;`

Quiz

```
package P is
   type T is limited null record;
   type R is record
      F1 : Integer;
      F2 : T;
   end record;
end P;

with P;
procedure Main is
   T1, T2 : P.T;
   R1, R2 : P.R;
begin
```

Which assignment(s) is (are) legal?

A. `T1    := T2;`
B. `R1    := R2;`
C. *`R1.F1 := R2.F1;`*
D. `R2.F2 := R2.F2;`

Explanations

A. T1 and T2 are `limited` types
B. R1 and R2 contain `limited` types so they are also `limited`
C. Theses components are not `limited` types
D. These components are of a `limited` type

Creating Values

Creating Values

- Initialization is not assignment (but looks like it)!
- Via **limited constructor functions**
  - Functions returning values of limited types
- Via an **aggregate**
  - *limited aggregate* when used for a `limited` type

```
type Spin_Lock is limited record
  Flag : Interfaces.Unsigned_8;
end record;
...
Mutex : Spin_Lock := (Flag => 0); -- limited aggregate
```

Limited Constructor Functions

- Allowed wherever limited aggregates are allowed
- More capable (can perform arbitrary computations)
- Necessary when limited type is also private
    - Users won't have visibility required to express aggregate contents

```
function F return Spin_Lock
is
begin
  ...
  return (Flag => 0);
end F;
```

Writing Limited Constructor Functions

- Remember - copying is not allowed

```
function F return Spin_Lock is
  Local_X : Spin_Lock;
begin
  ...
  return Local_X; -- this is a copy - not legal
   -- (also illegal because of pass-by-reference)
end F;
```

```
Global_X : Spin_Lock;
function F return Spin_Lock is
begin
  ...
  -- This is not legal staring with Ada2005
  return Global_X; -- this is a copy
end F;
```

"Built In-Place"

- Limited aggregates and functions, specifically
- No copying done by implementation
  - Values are constructed in situ

```
Mutex : Spin_Lock := (Flag => 0);
```

```
function F return Spin_Lock is
begin
  return (Flag => 0);
end F;
```

Quiz

```
type T is limited record
   I : Integer;
end record;
```

Which piece(s) of code is (are) a legal constructor for T?

A.
```
function F return T is
begin
  return T (I => 0);
end F;
```

B.
```
function F return T is
  Val : Integer := 0;
begin
  return (I => Val);
end F;
```

C.
```
function F return T is
  Ret : T := (I => 0);
begin
  return Ret;
end F;
```

D.
```
function F return T is
begin
  return (0);
end F;
```

Quiz

```
type T is limited record
   I : Integer;
end record;
```

Which piece(s) of code is (are) a legal constructor for T?

A.
```
function F return T is
begin
  return T (I => 0);
end F;
```

B.
```
function F return T is
  Val : Integer := 0;
begin
  return (I => Val);
end F;
```

C.
```
function F return T is
  Ret : T := (I => 0);
begin
  return Ret;
end F;
```

D.
```
function F return T is
begin
  return (0);
end F;
```

Quiz

```
package P is
   type T is limited record
      F1 : Integer;
      F2 : Character;
   end record;
   Zero : T := (0, ' ');
   One : constant T := (1, 'a');
   Two : T;
   function F return T;
end P;
```

Which is a correct completion of F?

A. `return (3, 'c');`

B. `Two := (2, 'b');`
   `return Two;`

C. `return One;`

D. `return Zero;`

Quiz

```
package P is
   type T is limited record
      F1 : Integer;
      F2 : Character;
   end record;
   Zero : T := (0, ' ');
   One : constant T := (1, 'a');
   Two : T;
   function F return T;
end P;
```

Which is a correct completion of F?

A. *`return (3, 'c');`*

B. `Two := (2, 'b');`
   `return Two;`

C. `return One;`

D. `return Zero;`

`A` contains an "in-place" return. The rest all rely on other objects, which would require an (illegal) copy.

Extended Return Statements

Function Extended Return Statements

- *Extended return*
- Result is expressed as an object
- More expressive than aggregates
- Handling of unconstrained types
- Syntax (simplified):

```
return identifier : subtype [:= expression];

return identifier : subtype
[do
   sequence_of_statements ...
 end return];
```

Extended Return Statements Example

```
--  Implicitly limited array
type Spin_Lock_Array (Positive range <>) of Spin_Lock;

function F return Spin_Lock_Array is
begin
  return Result : Spin_Lock_Array (1 .. 10) do
    ...
  end return;
end F;
```

Expression / Statements Are Optional

- Without sequence (returns default if any)

```
function F return Spin_Lock is
begin
  return Result : Spin_Lock;
end F;
```

- With sequence

```
function F return Spin_Lock is
  X : Interfaces.Unsigned_8;
begin
  --  compute X ...
  return Result : Spin_Lock := (Flag => X);
end F;
```

Statements Restrictions

- **No** nested extended return
- **Simple** return statement **allowed**
  - **Without** expression
  - Returns the value of the **declared object** immediately

```
function F return Spin_Lock is
begin
  return Result : Spin_Lock do
    if Set_Flag then
      Result.Flag := 1;
      return;  -- returns 'Result'
    end if;
    Result.Flag := 0;
  end return; -- Implicit return
end F;
```

Quiz

```
type T is limited record
   I : Integer;
end record;

function F return T is
begin
   -- F body...
end F;

O : T := F;
```

Which declaration(s) of `F` is (are) valid?

A. `return Return : T := (I => 1)`

B. `return Result : T`

C. `return Value := (others => 1)`

D.
```
return R : T do
   R.I := 1;
end return;
```

Quiz

```
type T is limited record
   I : Integer;
end record;

function F return T is
begin
   -- F body...
end F;

O : T := F;
```

Which declaration(s) of `F` is (are) valid?

A. `return Return : T := (I => 1)`

B. *`return Result : T`*

C. `return Value := (others => 1)`

D.
```
return R : T do
   R.I := 1;
end return;
```

A. Using **`return`** reserved keyword
B. OK, default value
C. Extended return must specify type
D. OK

Combining Limited and Private Views

Limited Private Types

- A combination of **limited** and **private** views
  - No client compile-time visibility to representation
  - No client assignment or predefined equality
- The typical design idiom for **limited** types
- Syntax
  - Additional reserved word **limited** added to **private** type declaration

```
type defining_identifier is limited private;
```

Limited Private Type Rationale (1)

```
package Multiprocessor_Mutex is
  -- copying is prevented
  type Spin_Lock is limited record
    -- but users can see this!
    Flag : Interfaces.Unsigned_8;
  end record;
  procedure Lock (This : in out Spin_Lock);
  procedure Unlock (This : in out Spin_Lock);
  pragma Inline (Lock, Unlock);
end Multiprocessor_Mutex;
```

Limited Private Type Rationale (2)

```
package MultiProcessor_Mutex is
  -- copying is prevented AND users cannot see contents
  type Spin_Lock is limited private;
  procedure Lock (The_Lock : in out Spin_Lock);
  procedure Unlock (The_Lock : in out Spin_Lock);
  pragma Inline (Lock, Unlock);
private
  type Spin_Lock is ...
end MultiProcessor_Mutex;
```

Limited Private Type Completions

- Clients have the partial view as **limited** and **private**
- The full view completion can be any kind of type
- Not required to be a record type just because the partial view is limited

```
package P is
  type Unique_ID_T is limited private;
  ...
private
  type Unique_ID_T is range 1 .. 10;
end P;
```

Write-Only Register Example

```
package Write_Only is
  type Byte is limited private;
  type Word is limited private;
  type Longword is limited private;
  procedure Assign (Input : in Unsigned_8;
                    To    : in out Byte);
  procedure Assign (Input : in Unsigned_16;
                    To    : in out Word);
  procedure Assign (Input : in Unsigned_32;
                    To    : in out Longword);
private
  type Byte is new Unsigned_8;
  type Word is new Unsigned_16;
  type Longword is new Unsigned_32;
end Write_Only;
```

Explicitly Limited Completions

- Completion in Full view includes word **limited**
- Optional
- Requires a record type as the completion

```
package MultiProcessor_Mutex is
  type Spin_Lock is limited private;
  procedure Lock (This : in out Spin_Lock);
  procedure Unlock (This : in out Spin_Lock);
private
  type Spin_Lock is limited -- full view is limited as well
    record
      Flag : Interfaces.Unsigned_8;
    end record;
end MultiProcessor_Mutex;
```

Effects of Explicitly Limited Completions

- Allows no internal copying too
- Forces parameters to be passed by-reference

```
package MultiProcessor_Mutex is
  type Spin_Lock is limited private;
  procedure Lock (This : in out Spin_Lock);
  procedure Unlock (This : in out Spin_Lock);
private
  type Spin_Lock is limited record
    Flag : Interfaces.Unsigned_8;
  end record;
end MultiProcessor_Mutex;
```

Automatically Limited Full View

- When other limited types are used in the representation
- Recall composite types containing limited types are **limited** too

```
package Foo is
   type Legal is limited private;
   type Also_Legal is limited private;
   type Not_Legal is private;
   type Also_Not_Legal is private;
private
   type Legal is record
      S : A_Limited_Type;
   end record;
   type Also_Legal is limited record
      S : A_Limited_Type;
   end record;
   type Not_Legal is limited record
      S : A_Limited_Type;
   end record;
   type Also_Not_Legal is record
      S : A_Limited_Type;
   end record;
end Foo;
```

Quiz

```
package P is
   type Priv is private;
private
   type Lim is limited null record;
   -- Complete Here
end P;
```

Which of the following piece(s) of code is (are) legal?

A.
```
type Priv is record
  E : Lim;
end record;
```

B.
```
type Priv is record
  E : Float;
end record;
```

C.
```
type A is array (1 .. 10) of Lim;
type Priv is record
  F : A;
end record;
```

D.
```
type Priv is record
  Component : Integer := Lim'Size;
end record;
```

Quiz

```
package P is
   type Priv is private;
private
   type Lim is limited null record;
   -- Complete Here
end P;
```

Which of the following piece(s) of code is (are) legal?

A.
```
type Priv is record
  E : Lim;
end record;
```

B.
```
type Priv is record
  E : Float;
end record;
```

C.
```
type A is array (1 .. 10) of Lim;
type Priv is record
  F : A;
end record;
```

D.
```
type Priv is record
  Component : Integer := Lim'Size;
end record;
```

A. E has limited type, partial view of Priv must be **limited private**

B. F has limited type, partial view of Priv must be **limited private**

Quiz

```
package P is
  type L1_T is limited private;
  type L2_T is limited private;
  type P1_T is private;
  type P2_T is private;
private
  type L1_T is limited record
     Component : Integer;
  end record;
  type L2_T is record
     Component : Integer;
  end record;
  type P1_T is limited record
     Component : L1_T;
  end record;
  type P2_T is record
     Component : L2_T;
  end record;
end P;
```

What will happen when the above code is compiled?

A. Type P1_T will generate a compile error
B. Type P2_T will generate a compile error
C. Both type P1_T and type P2_T will generate compile errors
D. The code will compile successfully

Quiz

```
package P is
  type L1_T is limited private;
  type L2_T is limited private;
  type P1_T is private;
  type P2_T is private;
private
  type L1_T is limited record
     Component : Integer;
  end record;
  type L2_T is record
     Component : Integer;
  end record;
  type P1_T is limited record
     Component : L1_T;
  end record;
  type P2_T is record
     Component : L2_T;
  end record;
end P;
```

What will happen when the above code is compiled?

A. ***Type P1_T will generate a compile error***
B. Type P2_T will generate a compile error
C. Both type P1_T and type P2_T will generate compile errors
D. The code will compile successfully

Full definition of `P1_T` adds restrictions, which is not allowed. `P2_T` contains a component whose visible view is `limited`, the internal view is not `limited` so `P2_T` is not `limited`.

Lab

Limited Types Lab

- Requirements
  - Create an employee record data type consisting of a name, ID, hourly pay rate
    - ID should be a unique value generated for every record
  - Create a timecard record data type consisting of an employee record, hours worked, and total pay
  - Create a main program that generates timecards and prints their contents
- Hints
  - If the ID is unique, that means we cannot copy employee records

Limited Types Lab Solution - Employee Data (Spec)

```
package Employee_Data is

   subtype Name_T is String (1 .. 6);
   type Employee_T is limited private;
   type Hourly_Rate_T is delta 0.01 digits 6 range 0.0 .. 999.99;
   type Id_T is range 999 .. 9_999;

   function Create (Name : Name_T;
                    Rate : Hourly_Rate_T := 0.0)
                    return Employee_T;
   function Id (Employee : Employee_T)
                return Id_T;
   function Name (Employee : Employee_T)
                  return Name_T;
   function Rate (Employee : Employee_T)
                  return Hourly_Rate_T;

private
   type Employee_T is limited record
      Name : Name_T        := (others => ' ');
      Rate : Hourly_Rate_T := 0.0;
      Id   : Id_T          := Id_T'First;
   end record;
end Employee_Data;
```

Limited Types Lab Solution - Timecards (Spec)

```
with Employee_Data;
package Timecards is

   type Hours_Worked_T is digits 3 range 0.0 .. 24.0;
   type Pay_T is digits 6;
   type Timecard_T is limited private;

   function Create (Name  : Employee_Data.Name_T;
                    Rate  : Employee_Data.Hourly_Rate_T;
                    Hours : Hours_Worked_T)
                    return Timecard_T;

   function Id (Timecard : Timecard_T)
                return Employee_Data.Id_T;
   function Name (Timecard : Timecard_T)
                return Employee_Data.Name_T;
   function Rate (Timecard : Timecard_T)
                return Employee_Data.Hourly_Rate_T;
   function Pay (Timecard : Timecard_T)
                return Pay_T;
   function Image (Timecard : Timecard_T)
                return String;

private
   type Timecard_T is limited record
      Employee     : Employee_Data.Employee_T;
      Hours_Worked : Hours_Worked_T := 0.0;
      Pay          : Pay_T          := 0.0;
   end record;
end Timecards;
```

Limited Types Lab Solution - Employee Data (Body)

```
package body Employee_Data is

   Last_Used_Id : Id_T := Id_T'First;

   function Create (Name : Name_T;
                    Rate : Hourly_Rate_T := 0.0)
                    return Employee_T is
   begin
      return Ret_Val : Employee_T do
         Last_Used_Id := Id_T'Succ (Last_Used_Id);
         Ret_Val.Name := Name;
         Ret_Val.Rate := Rate;
         Ret_Val.Id   := Last_Used_Id;
      end return;
   end Create;

   function Id (Employee : Employee_T) return Id_T is
      (Employee.Id);
   function Name (Employee : Employee_T) return Name_T is
      (Employee.Name);
   function Rate (Employee : Employee_T) return Hourly_Rate_T is
      (Employee.Rate);

end Employee_Data;
```

Limited Types Lab Solution - Timecards (Body)

```
package body Timecards is

   function Create (Name  : Employee_Data.Name_T;
                    Rate  : Employee_Data.Hourly_Rate_T;
                    Hours : Hours_Worked_T)
                    return Timecard_T is
   begin
      return
        (Employee     => Employee_Data.Create (Name, Rate),
         Hours_Worked => Hours,
         Pay          => Pay_T (Hours) * Pay_T (Rate));
   end Create;

   function Id (Timecard : Timecard_T) return Employee_Data.Id_T is
      (Employee_Data.Id (Timecard.Employee));
   function Name (Timecard : Timecard_T) return Employee_Data.Name_T is
      (Employee_Data.Name (Timecard.Employee));
   function Rate (Timecard : Timecard_T) return Employee_Data.Hourly_Rate_T is
     (Employee_Data.Rate (Timecard.Employee));
   function Pay (Timecard : Timecard_T) return Pay_T is
      (Timecard.Pay);

   function Image
     (Timecard : Timecard_T)
      return String is
      Name_S : constant String := Name (Timecard);
      Id_S   : constant String :=
        Employee_Data.Id_T'Image (Employee_Data.Id (Timecard.Employee));
      Rate_S : constant String :=
        Employee_Data.Hourly_Rate_T'Image
          (Employee_Data.Rate (Timecard.Employee));
      Hours_S : constant String :=
        Hours_Worked_T'Image (Timecard.Hours_Worked);
      Pay_S : constant String := Pay_T'Image (Timecard.Pay);
   begin
      return
        Name_S & " (" & Id_S & ") => " & Hours_S & " hours * " & Rate_S &
        "/hour = " & Pay_S;
   end Image;
end Timecards;
```

Limited Types Lab Solution - Main

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

   One : constant Timecards.Timecard_T := Timecards.Create
        (Name  => "Fred  ",
         Rate  => 1.1,
         Hours => 2.2);
   Two : constant Timecards.Timecard_T := Timecards.Create
        (Name  => "Barney",
         Rate  => 3.3,
         Hours => 4.4);

begin
   Put_Line (Timecards.Image (One));
   Put_Line (Timecards.Image (Two));
end Main;
```

Summary

Summary

- Limited view protects against improper operations
  - Incorrect equality semantics
  - Copying via assignment
- Enclosing composite types are `limited` too
  - Even if they don't use keyword `limited` themselves
- Limited types are always passed by-reference
- Extended return statements work for any type
  - Ada 2005 and later
- Don't make types `limited` unless necessary
  - Users generally expect assignment to be available

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 or subprograms

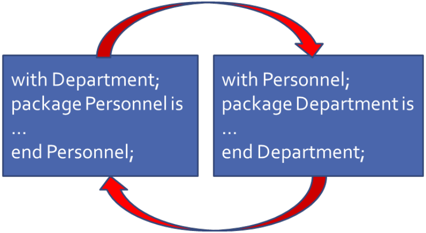
Library Units Review

- 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

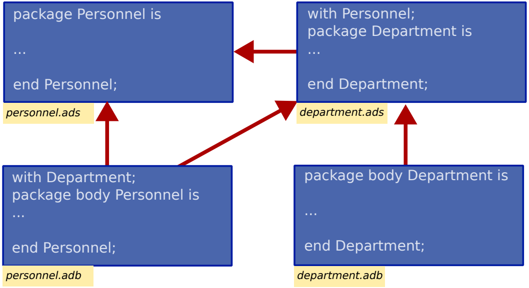
Handling Cyclic Dependencies

- 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

Circular Dependency in Package Declaration

```
with Department; --  Circular dependency
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; --  Circular dependency
package Department is
  type Section is private;
  procedure Choose_Manager (This : in out Section;
                            Who : in Personnel.Employee);
[...]
end Department;
```

limited with Clauses

- 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

- A type is *incomplete* when its representation is completely unknown
  - Address can still be manipulated through an `access`
  - Can be a formal parameter or function result's type
    - Subprogram's completion needs the complete type
    - Actual parameter needs the complete type
  - Can be a generic formal type parameters
  - If `tagged`, may also use **'Class**

```
type T;
```

- Can be declared in a **private** part of a package
  - And completed in its body
  - Used to implement opaque pointers
- Thus typically involves some advanced features

Legal Package Declaration Dependency

```ada
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;

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

- 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

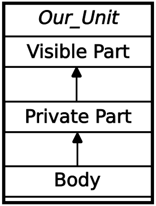
- Extensibility is a problem for private types
  - Provide excellent encapsulation and abstraction
  - But one has either complete visibility or essentially none
  - New functionality must be added to same package for sake of compile-time visibility to representation
  - Thus enhancements require editing/recompilation/retesting
- 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

Visibility Across a Hierarchy

In a **package**, the **body** sees everything the **private part** sees, and the **private part** sees everything the **visible part** sees.



Visibility Across a Hierarchy

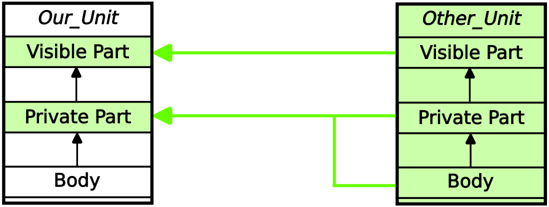
In a **package**, the **body** sees everything the **private part** sees, and the **private part** sees everything the **visible part** sees.

Another **package** can see our **visible part** (depending on where the "with" is), but nothing else.

Visibility Across a Hierarchy

In a **package**, the **body** sees everything the **private part** sees, and the **private part** sees everything the **visible part** sees.

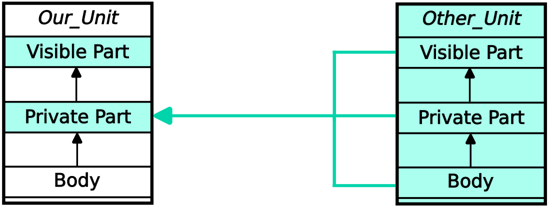
Our **child**'s **visible part** can see our **visible part**, and its **private part** (and **body**) can see our **private part**



Visibility Across a Hierarchy

In a **package**, the **body** sees everything the **private part** sees, and the **private part** sees everything the **visible part** sees.

Our **private child** can see our private part and **visible part** from anywhere



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 (are) **NOT** legal initialization(s) of `Child_Object`?

A. `Parent.Parent_Object + Parent.Sibling.Sibling_Object`
B. `Parent_Object + Sibling.Sibling_Object`
C. `Parent_Object + Sibling_Object`
D. `None 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 (are) **NOT** legal initialization(s) of `Child_Object`?

A. *`Parent.Parent_Object + Parent.Sibling.Sibling_Object`*
B. *`Parent_Object + Sibling.Sibling_Object`*
C. *`Parent_Object + Sibling_Object`*
D. `None 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
   Object_A : Integer;
private
   Object_B : Integer;
   procedure Dummy_For_Body;
end P;

package body P is
   Object_C : Integer;
   procedure Dummy_For_Body is null;
end P;

package P.Child is
   function X return Integer;
end P.Child;
```

Which return statement would 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
   Object_A : Integer;
private
   Object_B : Integer;
   procedure Dummy_For_Body;
end P;

package body P is
   Object_C : Integer;
   procedure Dummy_For_Body is null;
end P;

package P.Child is
   function X return Integer;
end P.Child;
```

Which return statement would 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

- 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

```
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

- Cyclic `limited with` clauses 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

```ada
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
- 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

- **use** Pkg; provides direct visibility into public items in Pkg
  - *Direct Visibility* - as if object was referenced from within package being used
  - *Public Items* - any entity defined in package spec public section
- 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" and "use all type" Clauses

"use type" and "use all type"

- `use type` makes **primitive operators** directly visible for specified type
  - Implicit and explicit operator function declarations

```
use type subtype_mark {, subtype_mark};
```

- `use all type` makes primitive operators **and all other operations** directly visible for specified type
  - All **enumerated type values** will also be directly visible

```
use all type subtype_mark {, subtype_mark};
```

- More specific alternatives to `use` clauses
  - Especially useful when multiple `use` clauses introduce ambiguity

Example Code

```ada
package Types is
  type Distance_T is range 0 .. Integer'Last;

  -- explicit declaration
  -- (we don't want a negative distance)
  function "-" (Left, Right : Distance_T)
              return Distance_T;

  -- implicit declarations (we get the division operator
  -- for "free", showing it for completeness)
  -- function "/" (Left, Right : Distance_T) return
  --              Distance_T;

  -- primitive operation
  function Min (A, B : Distance_T)
              return Distance_T;

end Types;
```

"use" Clauses Comparison

Blue = context clause being used

Red = compile errors with the context clause

No "use" clause

```
with Get_Distance;
with Types;
package Example is
  -- no context clause

  Point0 : Distance_T := Get_Distance;
  Point1 : Types.Distance_T := Get_Distance;
  Point2 : Types.Distance_T := Get_Distance;
  Point3 : Types.Distance_T := (Point1 - Point2) / 2;
  Point4 : Types.Distance_T := Min (Point1, Point2);

end Example;
```

"use" clause

```
with Get_Distance;
with Types;
package Example is
  use Types;

  Point0 : Distance_T := Get_Distance;
  Point1 : Types.Distance_T := Get_Distance;
  Point2 : Types.Distance_T := Get_Distance;
  Point3 : Types.Distance_T := (Point1 - Point2) / 2;
  Point4 : Types.Distance_T := Min (Point1, Point2);

end Example;
```

"use type" clause

```
with Get_Distance;
with Types;
package Example is
  use type Types.Distance;

  Point0 : Distance_T := Get_Distance;
  Point1 : Types.Distance_T := Get_Distance;
  Point2 : Types.Distance_T := Get_Distance;
  Point3 : Types.Distance_T := (Point1 - Point2) / 2;
  Point4 : Types.Distance_T := Min (Point1, Point2);

end Example;
```

"use all type" clause

```
with Get_Distance;
with Types;
package Example is
  use all type Types.Distance;

  Point0 : Distance_T := Get_Distance;
  Point1 : Types.Distance_T := Get_Distance;
  Point2 : Types.Distance_T := Get_Distance;
  Point3 : Types.Distance_T := (Point1 - Point2) / 2;
  Point4 : Types.Distance_T := Min (Point1, Point2);

end Example;
```

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;
```

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

- **renames** declaration creates an alias to an entity
  - 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

- `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
- `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

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 := Left;
begin
   Left := Right;
   Right := V;
end Swap_Int;

procedure Swap_Bool (Left, Right : in out Boolean) is
   V : Boolean := Left;
begin
   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

```
procedure Swap (Left, Right : in out (Integer | Boolean)) is
  V : (Integer | Boolean) := Left;
begin
   Left := Right;
   Right := V;
end Swap;
```

Solution: Generics

- A *generic unit* is a unit that does not exist
- It is a pattern based on properties
- The instantiation applies the pattern to certain parameters

Ada Generic Compared to C++ Template

Ada Generic

```
-- specification
generic
  type T is private;
procedure Swap (L, R : in out T);

-- implementation
procedure Swap (L, R : in out T) is
   Tmp : T := L;
begin
   L := R;
   R := Tmp;
end Swap;

-- instance
procedure Swap_F is new Swap (Float);
```

C++ Template

```
// prototype
template <class T>
void Swap (T & L, T & R);

// implementation
template <class T>
void Swap (T & L, T & R) {
   T Tmp = L;
   L = R;
   R = Tmp;
}

// instance
int x, y;
Swap<int>(x,y);
```

Creating Generics

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**
- Children of generic units have to be generic themselves

```
generic
package Stack.Utilities is
   procedure Print (S : Stack_T);
```

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

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.

Generic Data

Generic Types Parameters (1/3)

- A generic parameter is a template
- It specifies the properties the generic body can rely on

```
generic
   type T1 is private;
   type T2 (<>) is private;
   type T3 is limited private;
package Parent is
```

- The actual parameter must be no more restrictive then the *generic contract*

Generic Types Parameters (2/3)

- Generic formal parameter tells generic what it is allowed to do with the type

`type T1 is (<>);`	Discrete type; `'First`, `'Succ`, etc available
`type T2 is range <>;`	Signed Integer type; appropriate mathematic operations allowed
`type T3 is digits <>;`	Floating point type; appropriate mathematic operations allowed
`type T4;`	Incomplete type; can only be used as target of `access`
`type T5 is tagged private;`	`tagged` type; can extend the type
`type T6 is private;`	No knowledge about the type other than assignment, comparison, object creation allowed
`type T7 (<>) is private;`	`(<>)` indicates type can be unconstrained, so any object has to be initialized

Generic Types Parameters (3/3)

- The usage in the generic has to follow the contract
  - Generic Subprogram

```
generic
   type T (<>) is private;
procedure P (V : T);
procedure P (V : T) is
   X1 : T := V; -- OK, can constrain by initialization
   X2 : T;      -- Compilation error, no constraint to this
begin
```

  - Instantiations

```
type Limited_T is limited null record;

-- unconstrained types are accepted
procedure P1 is new P (String);

-- type is already constrained
-- (but generic will still always initialize objects)
procedure P2 is new P (Integer);

-- Illegal: the type can't be limited because the generic
-- thinks it can make copies
procedure P3 is new P (Limited_T);
```

Generic Parameters Can Be Combined

- Consistency is checked at compile-time

```
generic
   type T (<>) is private;
   type Acc is access all T;
   type Index is (<>);
   type Arr is array (Index range <>) of Acc;
function Element (Source   : Arr;
                  Position : Index)
                  return T;

type String_Ptr is access all String;
type String_Array is array (Integer range <>)
    of String_Ptr;

function String_Element is new Element
   (T     => String,
    Acc   => String_Ptr,
    Index => Integer,
    Arr   => String_Array);
```

Quiz

```
generic
   type T1 is (<>);
   type T2 (<>) is private;
procedure G
  (A : T1;
   B : T2);
```

Which is (are) legal instantiation(s)?

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 (are) legal instantiation(s)?

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/Variables As 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 package

```
generic
  type Element_T is private;
  Array_Size     : Positive;
  High_Watermark : in out Element_T;
package Repository is
```

- Generic instance

```
V   : Float;
Max : Float;

procedure My_Repository is new Repository
  (Element_T      => Float,
   Array_size     => 10,
   High_Watermark => Max);
```

Generic Subprogram Parameters

- Subprograms can be defined in the generic contract
- Must be introduced by **with** to differ from the generic unit

```
generic
   type T is private;
   with function Less_Than (L, R : T) return Boolean;
function Max (L, R : T) return T;

function Max (L, R : T) return T is
begin
   if Less_Than (L, R) then
      return R;
   else
      return L;
   end if;
end Max;

type Something_T is null record;
function Less_Than (L, R : Something_T) return Boolean;
procedure My_Max is new Max (Something_T, Less_Than);
```

Generic Subprogram Parameters Defaults

- **is** <> - matching subprogram is taken by default
- **is null** - null procedure is taken by default
  - Only available in Ada 2005 and later

```
generic
  type T is private;
  with function Is_Valid (P : T) return Boolean is <>;
  with procedure Error_Message (P : T) is null;
procedure Validate (P : T);

function Is_Valid_Record (P : Record_T) return Boolean;

procedure My_Validate is new Validate (Record_T,
                                       Is_Valid_Record);
-- Is_Valid maps to Is_Valid_Record
-- Error_Message maps to a null procedure
```

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 (Numeric))`
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 (Numeric))`
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

Given the following generic function:

```
generic
   type Some_T is private;
   with function "+" (L : Some_T; R : Integer) return Some_T is <>;
function Incr (Param : Some_T) return Some_T;

function Incr (Param : Some_T) return Some_T is
begin
   return Param + 1;
end Incr;
```

And the following declarations:

```
type Record_T is record
   Component : Integer;
end record;
function Add (L : Record_T; I : Integer) return Record_T is
   ((Component => L.Component + I))
function Weird (L : Integer; R : Integer) return Integer is (0);
```

Which of the following instantiation(s) is/are **not** legal?

A. `function IncrA is new Incr (Integer, Weird);`
B. `function IncrB is new Incr (Record_T, Add);`
C. `function IncrC is new Incr (Record_T);`
D. `function IncrD is new Incr (Integer);`

Quiz

Given the following generic function:

```
generic
   type Some_T is private;
   with function "+" (L : Some_T; R : Integer) return Some_T is <>;
function Incr (Param : Some_T) return Some_T;

function Incr (Param : Some_T) return Some_T is
begin
   return Param + 1;
end Incr;
```

And the following declarations:

```
type Record_T is record
   Component : Integer;
end record;
function Add (L : Record_T; I : Integer) return Record_T is
   ((Component => L.Component + I))
function Weird (L : Integer; R : Integer) return Integer is (0);
```

Which of the following instantiation(s) is/are **not** legal?

A. `function IncrA is new Incr (Integer, Weird);`
B. `function IncrB is new Incr (Record_T, Add);`
C. *`function IncrC is new Incr (Record_T);`*
D. `function IncrD is new Incr (Integer);`

`with function "+" (L : Some_T; R : Integer) return Some_T is <>;` indicates that if no function for + is passed in, find (if possible) a matching definition at the point of instantiation.

A. `Weird` matches the subprogram profile, so `Incr` will use `Weird` when doing addition for `Integer`
B. `Add` matches the subprogram profile, so `Incr` will use `Add` when doing the addition for `Record_T`
C. There is no matching + operation for `Record_T`, so that instantiation fails to compile
D. Because there is no parameter for the generic formal parameter +, the compiler will look for one in the scope of the instantiation. Because the instantiating type is numeric, the inherited + operator is found

Generic Completion

Implications at Compile-Time

- The body needs to be visible when compiling the user code
- Therefore, when distributing a component with generics to be instantiated, the code of the generic must come along

Generic and Freezing Points

- A generic type **freezes** the type and needs the **full view**
- May force separation between its declaration (in spec) and instantiations (in private or body)

```
generic
   type X is private;
package Base is
   V : access X;
end Base;

package P is
   type X is private;
   -- illegal
   package B is new Base (X);
private
   type X is null record;
end P;
```

Generic Incomplete Parameters

- A generic type can be incomplete
- Allows generic instantiations before full type definition
- Restricts the possible usages (only `access`)

```
generic
   type X; -- incomplete
package Base is
   V : access X;
end Base;

package P is
   type X is private;
   -- legal
   package B is new Base (X);
private
   type X is null record;
end P;
```

Quiz

```
generic
   type T1;
   A1 : access T1;
   type T2 is private;
   A2, B2 : T2;
procedure G_P;
procedure G_P is
begin
   -- Complete here
end G_P;
```

Which of the following statement(s) is (are) legal for `G_P`'s body?

A. `pragma Assert (A1 /= null)`
B. `pragma Assert (A1.all'Size > 32)`
C. `pragma Assert (A2 = B2)`
D. `pragma Assert (A2 - B2 /= 0)`

Quiz

```
generic
   type T1;
   A1 : access T1;
   type T2 is private;
   A2, B2 : T2;
procedure G_P;
procedure G_P is
begin
   -- Complete here
end G_P;
```

Which of the following statement(s) is (are) legal for `G_P`'s body?

A. *`pragma Assert (A1 /= null)`*
B. `pragma Assert (A1.all'Size > 32)`
C. *`pragma Assert (A2 = B2)`*
D. `pragma Assert (A2 - B2 /= 0)`

Genericity Lab

- Requirements
  - Create a record structure containing multiple components
    - Need subprograms to convert the record to a string, and compare the order of two records
    - Lab prompt package `Data_Type` contains a framework
  - Create a generic list implementation
    - Need subprograms to add items to the list, sort the list, and print the list
  - The **main** program should:
    - Add many records to the list
    - Sort the list
    - Print the list
- Hints
  - Sort routine will need to know how to compare elements
  - Print routine will need to know how to print one element

Genericity Lab Solution - Generic (Spec)

```
generic
   type Element_T is private;
   Max_Size : Natural;
   with function ">" (L, R : Element_T) return Boolean is <>;
   with function Image (Element : Element_T) return String;
package Generic_List is

   type List_T is private;

   procedure Add (This : in out List_T;
                  Item : in     Element_T);
   procedure Sort (This : in out List_T);
   procedure Print (List : List_T);

private
   subtype Index_T is Natural range 0 .. Max_Size;
   type List_Array_T is array (1 .. Index_T'Last) of Element_T;

   type List_T is record
      Values : List_Array_T;
      Length : Index_T := 0;
   end record;
end Generic_List;
```

Genericity Lab Solution - Generic (Body)

```
with Ada.Text_io; use Ada.Text_IO;
package body Generic_List is

   procedure Add (This : in out List_T;
                  Item : in     Element_T) is
   begin
      This.Length               := This.Length + 1;
      This.Values (This.Length) := Item;
   end Add;

   procedure Sort (This : in out List_T) is
      Temp : Element_T;
   begin
      for I in 1 .. This.Length loop
         for J in 1 .. This.Length - I loop
            if This.Values (J) > This.Values (J + 1) then
               Temp                := This.Values (J);
               This.Values (J)     := This.Values (J + 1);
               This.Values (J + 1) := Temp;
            end if;
         end loop;
      end loop;
   end Sort;

   procedure Print (List : List_T) is
   begin
      for I in 1 .. List.Length loop
         Put_Line (Integer'Image (I) & ") " & Image (List.Values (I)));
      end loop;
   end Print;

end Generic_List;
```

Genericity Lab Solution - Main

```
with Data_Type;
with Generic_List;
procedure Main is
   package List is new Generic_List (Element_T => Data_Type.Record_T,
                                     Max_Size  => 20,
                                     ">"       => Data_Type.">",
                                     Image => Data_Type.Image);

   My_List : List.List_T;
   Element : Data_Type.Record_T;

begin
   List.Add (My_List, (Integer_Component   => 111,
                       Character_Component => 'a'));
   List.Add (My_List, (Integer_Component   => 111,
                       Character_Component => 'z'));
   List.Add (My_List, (Integer_Component   => 111,
                       Character_Component => 'A'));
   List.Add (My_List, (Integer_Component   => 999,
                       Character_Component => 'B'));
   List.Add (My_List, (Integer_Component   => 999,
                       Character_Component => 'Y'));
   List.Add (My_List, (Integer_Component   => 999,
                       Character_Component => 'b'));
   List.Add (My_List, (Integer_Component   => 112,
                       Character_Component => 'a'));
   List.Add (My_List, (Integer_Component   => 998,
                       Character_Component => 'z'));

   List.Sort (My_List);
   List.Print (My_List);
end Main;
```

Summary

Generic Routines Vs Common Routines

```
package Helper is
   type Float_T is digits 6;
   generic
      type Type_T is digits <>;
      Min : Type_T;
      Max : Type_T;
   function In_Range_Generic (X : Type_T) return Boolean;
   function In_Range_Common (X   : Float_T;
                             Min : Float_T;
                             Max : Float_T)
                             return Boolean;
end Helper;

procedure User is
  type Speed_T is new Float_T range 0.0 .. 100.0;
  B : Boolean;
  function Valid_Speed is new In_Range_Generic
     (Speed_T, Speed_T'First, Speed_T'Last);
begin
  B := Valid_Speed (12.3);
  B := In_Range_Common (12.3, Speed_T'First, Speed_T'Last);
```

Summary

- Generics are useful for copying code that works the same just for different types
  - Sorting, containers, etc
- Properly written generics only need to be tested once
  - But testing / debugging can be more difficult
- Generic instantiations are best done at compile time
  - At the package level
  - Can be run time expensive when done in subprogram scope

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 as part of the language definition
  - Range specifications
  - Parameter modes
  - Generic contracts
  - OOP `interface` types
  - Work well, but on a restricted set of use-cases
- Contract aspects to be more expressive
  - 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 newer versions of the language
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 newer versions of the language
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** were fully integrated into Ada 2012
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

`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 controlled 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 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

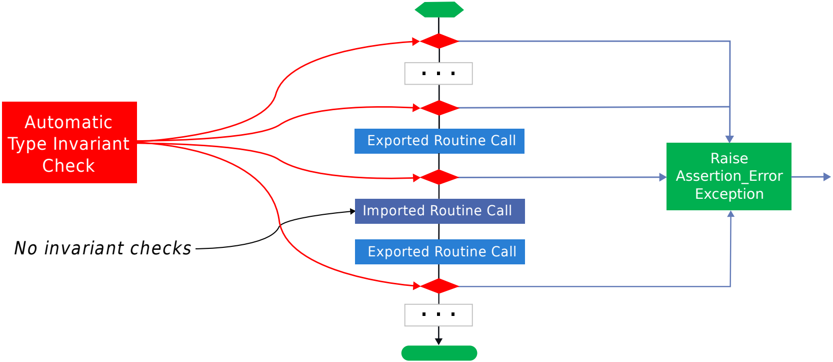
```
package Bank is
   type Account_T is private with Type_Invariant => Consistent_Balance (Account_T);
   type Currency_T is delta 0.01 digits 12;
   function Consistent_Balance (This : Account_T) return Boolean;
   procedure Open (This : in out Account_T; Initial_Deposit : Currency_T);
private
   type 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

- Applied to `private` types
- 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`
- Can apply to `type` or `subtype`

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;
```

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

Tasking

Introduction

Concurrency Mechanisms

- Task
  - **Active**
  - Rendezvous: **Client / Server** model
  - Server **entries**
  - Client **entry calls**
  - Typically maps to OS threads
- Protected object
  - **Passive**
  - *Monitors* protected data
  - **Restricted** set of operations
  - Concurrency-safe **semantics**
  - No thread overhead
  - Very portable
- Object-Oriented
  - `Synchronized` interfaces
  - Protected objects inheritance

A Simple Task

- Concurrent code execution via **task**
- `limited` types (No copies allowed)

```
procedure Main is
   task type Simple_Task_T;
   task body Simple_Task_T is
   begin
      loop
         delay 1.0;
         Put_Line ("T");
      end loop;
   end Simple_Task_T;
   Simple_Task : Simple_Task_T;
   -- This task starts when Simple_Task is elaborated
begin
   loop
      delay 1.0;
      Put_Line ("Main");
   end loop;
end;
```

- A task is started when its declaration scope is **elaborated**
- Its enclosing scope exits when **all tasks** have finished

Tasks

Rendezvous Definitions

- **Server** declares several `entry`
- Client calls entries like subprograms
- Server `accept` the client calls
- At each standalone `accept`, server task **blocks**
  - **Until** a client calls the related `entry`

```
task type Msg_Box_T is
   entry Start;
   entry Receive_Message (S : String);
end Msg_Box_T;

task body Msg_Box_T is
begin
   loop
      accept Start;
      Put_Line ("start");

      accept Receive_Message (S : String) do
         Put_Line ("receive " & S);
      end Receive_Message;
   end loop;
end Msg_Box_T;

T : Msg_Box_T;
```

Rendezvous Entry Calls

- Upon calling an `entry`, client **blocks**
  - **Until** server reaches `end` of its `accept` block

```
Put_Line ("calling start");
T.Start;
Put_Line ("calling receive 1");
T.Receive_Message ("1");
Put_Line ("calling receive 2");
T.Receive_Message ("2");
```

- May be executed as follows:

```
calling start
start              -- May switch place with line below
calling receive 1  -- May switch place with line above
receive 1
calling receive 2
-- Blocked until another task calls Start
```

Rendezvous with a Task

- `accept` statement
  - Wait on single entry
  - If entry call waiting: Server handles it
  - Else: Server **waits** for an entry call
- `select` statement
  - **Several** entries accepted at the **same time**
  - Can **time-out** on the wait
  - Can be **not blocking** if no entry call waiting
  - Can **terminate** if no clients can **possibly** make entry call
  - Can **conditionally** accept a rendezvous based on a **guard expression**

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: Functions and Procedures

- A `function` can **get** the state
  - **Multiple-Readers**
  - Protected data is **read-only**
  - Concurrent call to `function` is **allowed**
  - **No** concurrent call to `procedure`
- A `procedure` can **set** the state
  - **Single-Writer**
  - **No** concurrent call to either `procedure` or `function`
  - In case of concurrency, other callers get **blocked**
    - Until call finishes

Example

```
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;
```

Delays

Delay Keyword

- `delay` keyword part of tasking
- Blocks for a time
- Relative: Blocks for at least `Duration`
- Absolute: Blocks until no earlier than `Calendar.Time` or `Real_Time.Time`

```
with Calendar;

procedure Main is
   Relative : Duration := 1.0;
   Absolute : Calendar.Time
     := Calendar.Time_Of (2030, 10, 01);
begin
   delay Relative;
   delay until Absolute;
end Main;
```

Task and Protected Types

Task Activation

- Instantiated tasks start running when **activated**
- On the **stack**
  - When **enclosing** declarative part finishes **elaborating**
- On the **heap**
  - **Immediately** at instantiation

```
task type First_T is ...
type First_T_A is access all First_T;

task body First_T is ...
...
declare
   V1 : First_T;
   V2 : First_T_A;
begin  -- V1 is activated
   V2 := new First_T;  -- V2 is activated immediately
```

Single Declaration

- Instantiate an **anonymous** task (or protected) type
- Declares an object of that type

```
task type Task_T is
   entry Start;
end Task_T;

type Task_Ptr_T is access all Task_T;

task body Task_T is
begin
   accept Start;
end Task_T;
...
   V1 : Task_T;
   V2 : Task_Ptr_T;
begin
   V1.Start;
   V2 := new Task_T;
   V2.all.Start;
```

Task Scope

- Nesting is possible in **any** declarative block
- Scope has to **wait** for tasks to finish before ending
- At library level: program ends only when **all tasks** finish

```
package P is
   task type T;
end P;

package body P is
   task body T is
      loop
         delay 1.0;
         Put_Line ("tick");
      end loop;
   end T;

   Task_Instance : T;
end P;
```

Some Advanced Concepts

Waiting on Multiple Entries

- select can wait on multiple entries
  - With **equal** priority, regardless of declaration order

```
loop
  select
    accept Receive_Message (V : String)
    do
      Put_Line ("Message : " & V);
    end Receive_Message;
  or
    accept Stop;
    exit;
  end select;
end loop;
...
T.Receive_Message ("A");
T.Receive_Message ("B");
T.Stop;
```

Waiting with a Delay

- A **select** statement may **time-out** using **delay** or **delay until**
  - Resume execution at next statement
- Multiple **delay** allowed
  - Useful when the value is not hard-coded

```
loop
  select
    accept Receive_Message (V : String) do
      Put_Line ("Message : " & V);
    end Receive_Message;
  or
    delay 50.0;
    Put_Line ("Don't wait any longer");
    exit;
  end select;
end loop;
```

Task will wait up to 50 seconds for `Receive_Message`. *If no message is received, it will write to the console, and then restart the loop. (If the* `exit` *wasn't there, the loop would exit the first time no message was received.)*

Calling an Entry with a Delay Protection

- A call to `entry` **blocks** the task until the entry is `accept` 'ed
- Wait for a **given amount of time** with `select ... delay`
- Only **one** entry call is allowed
- No `accept` statement is allowed

```
task Msg_Box is
   entry Receive_Message (V : String);
end Msg_Box;

procedure Main is
begin
   select
      Msg_Box.Receive_Message ("A");
   or
      delay 50.0;
   end select;
end Main;
```

Procedure will wait up to 50 seconds for `Receive_Message` *to be accepted before it gives up*

Non-blocking Accept or Entry

- Using `else`
  - Task **skips** the `accept` or `entry` call if they are **not ready** to be entered
- `delay` is **not** allowed in this case

```
select
   accept Receive_Message (V : String) do
      Put_Line ("Received : " & V);
   end Receive_Message;
else
   Put_Line ("Nothing to receive");
end select;

[...]

select
   T.Receive_Message ("A");
else
   Put_Line ("Receive message not called");
end select;
```

Queue

- Protected `entry` or `procedure` and tasks `entry` are activated by **one** task at a time
- **Mutual exclusion** section
- Other tasks trying to enter are **queued**
  - In **First-In First-Out** (FIFO) by default
- When the server task **terminates**, tasks still queued receive `Tasking_Error`

Advanced Tasking

Other constructions are available

- **Guard condition** on `accept`
- `requeue` to **defer** handling of an `entry` call
- `terminate` the task when no `entry` call can happen anymore
- `abort` to stop a task immediately
- `select ... then abort` some other task

Lab

Tasking Lab

- Requirements
  - Create multiple tasks with the following attributes
    - Startup entry receives some identifying information and a delay length
    - Stop entry will end the task
    - Until stopped, the task will send it's identifying information to a monitor periodically based on the delay length
  - Create a protected object that stores the identifying information of task that called it
  - Main program should periodically check the protected object, and print when it detects a task switch
    - I.e. If the current task is different than the last printed task, print the identifying information for the current task

Tasking Lab Solution - Protected Object

```
with Task_Type;
package Protected_Object is
   protected Monitor is
      procedure Set (Id : Task_Type.Task_Id_T);
      function Get return Task_Type.Task_Id_T;
   private
      Value : Task_Type.Task_Id_T;
   end Monitor;
end Protected_Object;

package body Protected_Object is
   protected body Monitor is
      procedure Set (Id : Task_Type.Task_Id_T) is
      begin
         Value := Id;
      end Set;
      function Get return Task_Type.Task_Id_T is (Value);
   end Monitor;
end Protected_Object;
```

Tasking Lab Solution - Task Type

```
package Task_Type is
   type Task_Id_T is range 1_000 .. 9_999;
   task type Task_T is
      entry Start_Task (Task_Id        : Task_Id_T;
                        Delay_Duration : Duration);
      entry Stop_Task;
   end Task_T;
end Task_Type;

with Protected_Object;
package body Task_Type is
   task body Task_T is
      Wait_Time : Duration;
      Id        : Task_Id_T;
   begin
      accept Start_Task (Task_Id        : Task_Id_T;
                         Delay_Duration : Duration) do
         Wait_Time := Delay_Duration;
         Id        := Task_Id;
      end Start_Task;
      loop
         select
            accept Stop_Task;
            exit;
         or
            delay Wait_Time;
            Protected_Object.Monitor.Set (Id);
         end select;
      end loop;
   end Task_T;
end Task_Type;
```

Tasking Lab Solution - Main

```
with Ada.Text_IO; use Ada.Text_IO;
with Protected_Object;
with Task_Type;
procedure Main is
   T1, T2, T3       : Task_Type.Task_T;
   Last_Id, This_Id : Task_Type.Task_Id_T := Task_Type.Task_Id_T'Last;
   use type Task_Type.Task_Id_T;
begin

   T1.Start_Task (1_111, 0.3);
   T2.Start_Task (2_222, 0.5);
   T3.Start_Task (3_333, 0.7);

   for Count in 1 .. 20 loop
      This_Id := Protected_Object.Monitor.Get;
      if Last_Id /= This_Id then
         Last_Id := This_Id;
         Put_Line (Count'Image & "> " & Last_Id'image);
      end if;
      delay 0.2;
   end loop;

   T1.Stop_Task;
   T2.Stop_Task;
   T3.Stop_Task;

end Main;
```

Summary

Summary

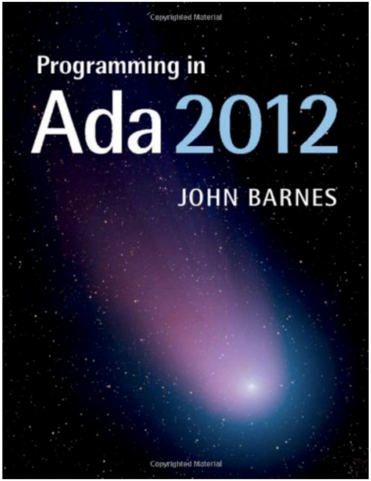
- Tasks are **language-based** concurrency mechanisms
    - Typically implemented as threads
    - Not necessarily for **truly** parallel operations
    - Originally for task-switching / time-slicing
- Multiple mechanisms to **synchronize** tasks
    - Delay
    - Rendezvous
    - Queues
    - Protected Objects

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!

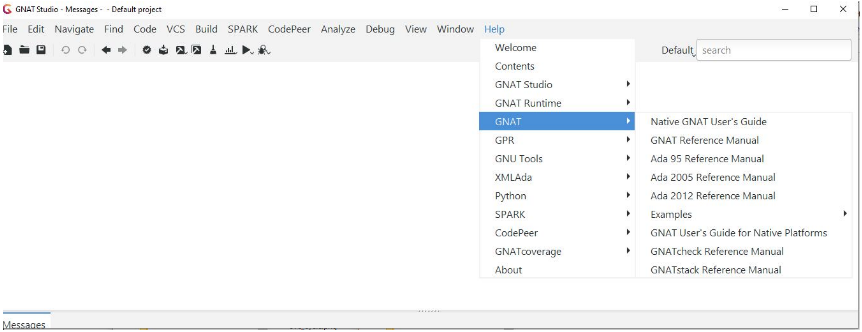
Current Ada Standard

- "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 "Case" via the GNATtracker web interface and/or email
  - GNATtracker
    - Select "Create A New Case" from the main landing page
  - Email
    - Send to: support@adacore.com
    - Subject should read: #XXXX - (descriptive text)
- Not just for "bug reports"
  - Ask questions, make suggestions, etc.