Introduction

About AdaCore

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

About This Training

Your Trainer

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

About This Training

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

I Note

This is an important piece of info

? Tip This is a tip

Overview

A Little History

A Little History

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

A Little History

Ada Evolution Highlights

Ada 83 Abstract Data Types Modules Concurrency Generics Exceptions

- Ada 2012 Contracts Iterators Flexible Expressions
- Ada 2022 'Image for all types Declare expression

- Ada 95 OOP Child Packages Annexes
- Ada 2005 Multiple Inheritance Containers Ravenscar

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)

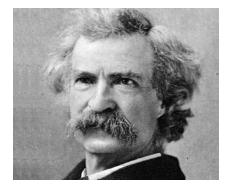
- Language-Based Concurrency
 - Explicit interactions
 - Run-time handling
 - Portable
- Low Level Programming
 - Define representation of types
 - Storage pools definition
 - Foreign language integration

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

Setup

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:

"Hello World" Lab - ${\rm GNAT}\ {\rm Studio}$

- 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

AdaCore

Setup

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

larations

Introduction

Introduction

Ada Type Model

■ Each *object* is associated with 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 an *identifier* to an *entity*
 - Objects
 - Types
 - Subprograms
 - et cetera
- In a *declarative part*
- Example: Something : Typemark := Value;
 - Something is an *identifier*
- **Some** implicit declarations
 - Standard types and operations
 - Implementation-defined

▲ Warning

Declaration must precede use

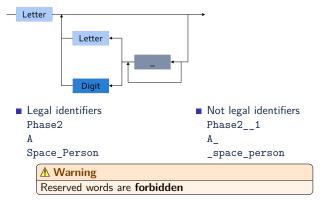
AdaCore

Identifiers and Comments

Identifiers and Comments

Declarations

Identifiers



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

AdaCore

Identifiers vs Names

identifier Syntactic form used typically to introduce entities when declared

name Typically starts with an identifier and can be followed by one or more suffixes to help indicate something more specific, such as a record component or an array slice

🖓 Tip

An **identifier** is used to *define* an entity, and a **name** is used to *refer to* an entity (or part of one)

Reserved Words

abort
abs
abstract (95)
accept
access
aliased (95)
all
and
array
at
begin
body
case
constant
declare
delay
delta
digits
do

else elsif end entry exception exit for function generic goto if in interface (2005) is limited loop mod new not

null of or others out overriding (2005) package parallel (2022) pragma private procedure protected (95) raise range record rem renames requeue (95) return

reverse select separate some (2012) subtype synchronized (2005) tagged (95) task terminate then type until (95) use when while with xor

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

Identifiers and Comments

Declaring Constants / Variables (simplified)

■ An <i>expressic</i>	<i>n</i> is a piece of Ada code that returns a value .
<identifier> :</identifier>	<pre>constant := <expression>;</expression></pre>
<identifier> :</identifier>	<type> := <expression>;</expression></type>
<identifier> :</identifier>	<pre>constant <type> := <expression>;</expression></type></pre>

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

Declarations			
Literals			

Literals

String Literals

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

Literals

Decimal Numeric Literals

Syntax

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

💡 Tip

Underscore is not significant and helpful for grouping

E (exponent) must always be integer

Examples

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

Literals

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_111# => 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

Literals

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;

Literals

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

- Inderscores 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
- ▶ Missing closing #

Object Declarations

- An object is either *variable* or *constant*
- Basic Syntax
 - <name> : <subtype> [:= <initial value>];
 - <name> : constant <subtype> := <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);
```

```
Declarations
```

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);

A 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** \rightarrow 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
```

AdaCore

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

AdaCore

Universal Types

Universal Types

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;

Universal Types

Numeric Literals Are Universally Typed

- No need to type them
 - e.g OUL as in C
- Compiler handles typing

Note

No bugs with precision

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

Literals Must Match "Class" of Context

- $\blacksquare universal_integer \ literals \rightarrow Integer$
- \blacksquare universal_real literals \rightarrow 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

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 Numbers

Named Number Benefit

Evaluation at compile time

As if used directly in the code

Tip Useful due to their **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;	3.3333333333333333333338E-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** \rightarrow *in scope* but not **directly** visible

Introducing Block Statements

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

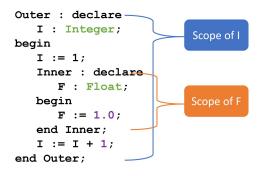
```
Syntax
[<block-name> :] declare
        <declarative part>
    begin
        <statements>
end [block-name];
```

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

Scope and "Lifetime"

\blacksquare Object in scope \rightarrow exists





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;
```

AdaCore

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)

```
1 declare
```

10

 $11 \\ 12$

```
M : Integer := 1;
2
    begin
3
       M := M + 1;
4
       declare
\mathbf{5}
           M : Integer := 2;
6
       begin
7
           M := M + 2;
8
           Print (M);
9
```

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)

```
1 declare
```

```
2 M : Integer := 1;
```

```
3 begin
```

```
M := M + 1;
4
       declare
           M : Integer := 2;
6
       begin
7
           M := M + 2:
8
           Print (M);
9
10
       end;
       Print (M);
11
12
    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

Declarations

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
```

AdaCore

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] }
```

I Note

Aspect clauses always part of a declaration

AdaCore

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 \rightarrow False

procedure Foo; -- Inline is False

Original form!

Summary

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)

AdaCore

Basic Types

	Types
Dasic	Types

Introduction

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

```
Basic Types
Introduction
```

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
AddCore
```

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

```
Basic Types
```

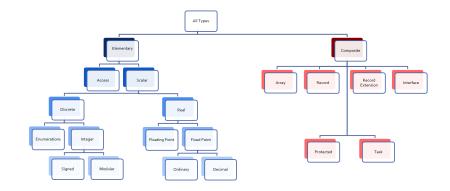
Introduction

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

Basic Types

Introduction

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);

AdaCore

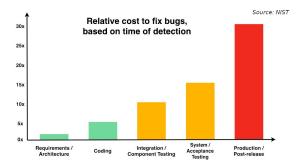
Type Model Run-Time Costs

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

I Note	
Compile-time check is free	
C	Ada
<pre>int X;</pre>	X : Integer;
int Y; // range 1 10	Y, Z : Integer range 1 10;
• • •	
if (X > 0 && X < 11)	Y := X;
Y = X;	Z := Y; no check required
else	
// signal a failure	

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
```

AdaCore

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

A Warning

Portability not guaranteed

But usage 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

I Note

Exponentiation (**) result will be a signed integer

So power must be Integer >= 0

▲ Warning

Division by zero \rightarrow 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 = (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 $T \rightarrow String$
- T'Value (input)
 - Converts String \rightarrow 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
     AdaCore
```

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 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

Unknown - depends on the compiler

Explanations

- 2¹⁰²⁴ 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

AdaCore

Enumeration Types

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 Types

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 := Mon;
Today := North; -- compile error
Heading := South;
Heading := East + 1; -- compile error
if Today < Tomorrow then ...</pre>
```

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



```
Basic Types
```

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
 - \blacksquare 0 \rightarrow none deployed, 0xF \rightarrow all deployed
- Then, use_deployed_inertia_matrix() if only first paddle is deployed!
- Better: boolean function paddles_deployed()
 - Single line to modify

AdaCore

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

- $\blacksquare \ Short-circuit \rightarrow fixed \ {\rm evaluation} \ order$
- 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?
```

Α.	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

Basic Types		
Real Types		

Real Types

Real Types

- Approximations to continuous values
 - **1**.0, 1.1, 1.11, 1.111 ... 2.0, ...
 - Finite hardware \rightarrow approximations
- Floating-point
 - Variable exponent
 - Large range
 - Constant relative precision
- Fixed-point
 - Constant exponent
 - Limited range
 - Constant absolute precision
 - Subdivided into Binary and Decimal
- Class focuses on floating-point

```
Basic Types
Real Types
```

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
 - $\blacksquare \ If none available \rightarrow declaration is \ \textbf{rejected}$
- 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

To keep portability

Floating Point Type Operators

By increasing precedence

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

Note

Exponentiation (**) result will be real

- So power must be Integer
 - Not possible to ask for root

X**0.5 \rightarrow 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

AdaCore

Numeric Types Conversion

Ada's integer and real are *numeric*

Holding a numeric value

Special rule: can always convert between numeric types

Explicitly

Marning $Float \rightarrow Integer causes rounding$

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;
```

7.6E-01
Compile Error
8.0E-01
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

Miscellaneous

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;
```

AdaCore

Basic Types	
Subtypes	

Subtypes

Subtype

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

```
subtype <identifier> is <type_name> [constraints];
```

Type_Name is an existing type or subtype

Note

If no constraint \rightarrow 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
 - $\blacksquare \text{ Initial value} \rightarrow \text{error}$
 - \blacksquare ... else \rightarrow warning

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

• • •

Max := 0; -- run-time error

AdaCore

```
Basic Types
Subtypes
```

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!
```

Subtypes

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 Days_Of_Week_T is (Sat, Sun, Mon, Tue, Wed, Thu, Fri);
subtype Weekdays_T is Days_Of_Week_T range Mon .. Fri;
```

Which subtype definition is valid?

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

Quiz

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

Which subtype definition is valid?

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

Explanations

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

AdaCore

Basic Types			
Lab			

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 \rightarrow 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

AdaCore

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;
1
   procedure Main is
2
3
      type Number_Of_Tests_T is range 0 .. 100;
4
      type Test Score Total T is digits 6 range 0.0 .. 10 000.0;
5
6
      type Degrees_T is mod 360;
7
8
      type Cymk T is (Cyan, Magenta, Yellow, Black);
9
10
      Number Of Tests : Number Of Tests T;
11
      Test_Score_Total : Test_Score_Total_T;
12
13
      Angle : Degrees T;
14
15
      Color : Cymk_T;
16
         AdaCore
                                                           137 / 928
```

Lab

Basic Types Lab Solution - Implementation

```
begin
18
19
      -- assignment
20
      Number Of Tests := 15;
21
      Test Score Total := 1 234.5;
22
      Angle := 180;
23
      Color
                      := Magenta;
24
25
      Put Line (Number_Of_Tests'Image);
26
      Put Line (Test Score Total'Image);
27
      Put Line (Angle'Image):
28
      Put Line (Color'Image):
20
30
      -- operations / attributes
31
      Test Score Total := Test Score Total / Test Score Total T (Number Of Tests);
32
      Angle := Angle + 359;
33
                      := Cvmk T'Pred (Cvmk T'Last);
      Color
34
35
      Put Line (Test Score Total'Image);
36
      Put_Line (Angle'Image);
37
      Put Line (Color'Image);
38
30
   end Main:
40
```

Summary

Benefits of Strongly Typed Numerics

Prevent subtle bugs

Basic Types Summary

- 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
 - \blacksquare More types \rightarrow more precision \rightarrow less bugs
 - Storing both feet and meters in Float has caused bugs
 - \blacksquare More types \rightarrow more complexity \rightarrow more bugs
 - A Green_Round_Object_Altitude type is probably never needed
- Default initialization is possible
 - Use sparingly

AdaCore

Statements

Introduction

Introduction

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

AdaCore

Procedure Calls (Overview)

Procedures must be defined before they are called

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

Block Statements

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

```
    Syntax
```

Block Statements

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

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

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

 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
```

Ada 2022

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

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 1 if Valve (N) /= Closed then
1
    Isolate (Valve (N));
                              2
                                   Isolate (Valve (N));
2
    Failure (Valve (N));
                                   Failure (Valve (N));
                              3
3
  else
                                 elsif System = Off then
4
                              4
    if System = Off then
                                   Failure (Valve (N));
                              5
5
      Failure (Valve (N)); 6 end if;
6
    end if;
7
  end if;
8
```

Case Statements

- Exclusionary choice among alternatives
- Syntax

```
case <expression> is
  when <choice> => <statements>;
  { when <choice> => <statements>; }
end case;
choice ::= <expression> | <discrete range>
```

```
| others { "|" <other choice> }
```

Simple "case" Statements

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

```
end case;
```

Note: No fall-through between cases

Conditional Statements

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:
```

Conditional Statements

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;
```

Conditional Statements

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

AdaCore

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
    When Sun =>
        Put_Line ("Day Off");
    when Mon | Fri =>
        Put_Line ("Short Day");
    when Tue .. Thu =>
        Put_Line ("Long Day");
    end case;
```

Conditional Statements

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
    When Sun =>
        Put_Line ("Day Off");
    when Mon | Fri =>
        Put_Line ("Short Day");
    when Tue .. Thu =>
        Put_Line ("Long Day");
    end case;
```

Explanations

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

```
AdaCore
```

Loop Statements

Loop Statements

```
Statements
```

Basic Loops and Syntax

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

Example

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



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;
```

```
Statements
```

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 loop -- anonymous subtype
-- Constant and variable range
for Day in Mon .. Fri loop
. . .
Today, Tomorrow : Days_T; -- assume some assignment...
for Day in Today .. Tomorrow loop
                                                  176 / 928
```

```
Statements
```

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
1
      type Color_T is (Red, White, Blue);
      type Rgb_T is (Red, Green, Blue);
3
4
   begin
      for Color in Red .. Blue loop -- which Red and Blue?
6
         null:
      end loop;
      for Color in Rgb_T'(Red) .. Blue loop -- OK
8
Q.
         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

```
Statements
```

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;
```

```
Statements
```

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;
```

AdaCore

```
Statements
```

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?
for A in 1 .. 10 loop
A := A + 1;
end loop;
for B in 1 .. 10 loop
Put_Line (Integer'Image (B));
end loop;
for C in reverse 1 .. 10 loop
Put_Line (Integer'Image (C));
end loop;
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
```

Cannot assign to a loop parameter

Legal - 10 iterations

C. Legal - 10 iterations

Legal - 0 iterations

GOTO Statements

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

AdaCore

189 / 928

Lab

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: 10 begin Dav Loop : for Day in Days_Of_Week_T loop Put Line (Day'Image); Input Loop : 100p 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 24 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: => Total Worked := Total Worked + Hours Today * 2.0; when Sunday end case; 32 end loop Day Loop; 34 Put Line (Total Worked'Image); 36 end Main;

AdaCo<u>re</u>

Summary

Summary,

Summary

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

Array Types

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

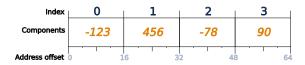
Index_Type

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);
```

AdaCore

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 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;
```

Constrained Array Types

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;
```

Constrained Array Types

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?
                                 Explanations
 A X1 (1) := Y1 (1):
                                     Boolean
 B. X1 := Y1;
 \bigcirc X1 (1) := X2 (1);
 D X2 := X1;
```

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

Unconstrained Array Types

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;

AdaCore

Unconstrained Array Types

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

AdaCore

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 components
- 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

AdaCore

Unconstrained Array Types

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"; Unconstrained Array Types

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

- An *indefinite type* does 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 components 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;
```

```
type Bit_T is range 0 .. 1;
type Bit_Array_T is array (Positive range <>) of Bit_T;
Which declaration(s) is (are)
legal?
AAA : Array_T (0..99);
```

- **B.** BBB : Array_T (1..32);
- C. CCC : Array_T (17..16);
- D. DDD : Array_T;

```
type Bit_T is range 0 .. 1;
type Bit_Array_T is array (Positive range <>) of Bit_T;
Which declaration(s) is (are) Explanations
legal? AAA : Array_T (0..99); which starts at 1
```

- B. OK, indices are in range
- OK, indicates a zero-length array
- D. Object must be constrained

B. BBB : Array_T (1..32);

D. DDD : Array_T;

C. CCC : Array T (17..16);

Array Types	
Attributes	

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

```
Array Types
Attributes
```

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
AdaCore
```

```
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
```

```
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
```

AdaCore

Operations

Operations

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

AdaCore

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));

```
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);
```

```
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

- All objects are just Boolean values
- B. A component 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

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 component
 - 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 components within an object
- Syntax hides indexing/iterator controls

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

- Starts with "first" component 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;
```

```
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
 0 10 15 20 0
 25 20 15 10 5
NB: Without Default Component Value, init. values are random
```

AdaCore

```
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?
                                Explanations
 A 1 10 15 20 1
                                  A There is a reverse
 B 1 20 15 10 1
                                  B. Yes
 0 10 15 20 0
                                  C Default value is 1
 D 25 20 15 10 5
                                  D No
```

NB: Without Default_Component_Value, init. values are random

AdaCore

Aggregates

Aggregates

Aggregates

Aggregates

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

AdaCore

. . .

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

"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

```
Array Types
Aggregates
```

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 => <>);

AdaCore

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?

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)

AdaCore

Aggregates in Ada 2022



```
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 component array
Ada2012 : Array_T := (1 => 5);
Illegal : Array_T := (5);
Ada2022 : Array T := [5];
```

Ada 2022

Iterated Component Association

■ With Ada 2022, we can create aggregates with *iterators*

- Basically, an inline looping mechanism
- Index-based iterator

- 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 component doubled

AdaCore

Ada 2022

More Information on Iterators



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

AdaCore

Delta Aggregates



```
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

Detour - 'Image for Complex Types

'Image Attribute



```
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]
```

AdaCore

Overriding the 'Image Attribute



- 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



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;
```

AdaCore

Using the 'Image Attribute

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

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

AdaCore



Ada 2022

Anonymous Array Types

Anonymous Array Types

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;

Array Types			
Lab			

Lab

Array Lab

Requirements

- Create an array type whose index is days of the week and each component 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 component 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;
1
   procedure Main is
2
3
      type Days Of Week T is
4
          (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
5
      type Unconstrained_Array_T is
6
         array (Days_Of_Week_T range <>) of Natural;
7
8
      Const_Arr : constant Unconstrained_Array_T := (1, 2, 3, 4
9
      Array_Var : Unconstrained_Array_T (Days_Of_Week_T);
10
11
      type Name_T is array (1 .. 6) of Character;
12
      type Names T is array (1 .. 3) of Name T;
13
      Weekly Staff : array (Days Of Week T) of Names T;
14
```

Array Lab Solution - Implementation

```
15 begin
      Array Var := Const Arr;
      for Item of Array Var loop
         Put Line (Item'Image);
18
      end loop;
19
      New Line;
20
21
22
      Array Var :=
        (Mon => 111, Tue => 222, Wed => 333, Thu => 444, Fri => 555, Sat => 666,
23
         Sun => 777):
24
      for Index in Array Var'Range loop
25
         Put Line (Index'Image & " => " & Array Var (Index)'Image);
26
      end loop:
27
      New Line:
28
      Array Var (Mon .. Wed) := Const Arr (Wed .. Fri);
30
      Array Var (Wed .. Fri) := (others => Natural'First);
31
      for Item of Array Var loop
         Put Line (Item'Image);
33
34
      end loop;
      New Line;
35
36
      Weekly Staff := (Mon | Tue | Thu | Fri => ("Fred ", "Barney", "Wilma "),
37
                            => ("closed", "closed", "closed"),
38
                        Wed
                        others => ("Pinky ", "Inky ", "Blinky"));
-40
41
      for Day in Weekly Staff'Range loop
         Put_Line (Day'Image);
42
         for Staff of Weekly Staff(Day) loop
            Put Line (" " & String (Staff));
         end loop;
46
      end loop;
47 end Main;
```

Summary

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

AdaCore

265 / 928

Record Types

Record Types		
Introduction		

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;
```

AdaCore

Components Rules

Components Rules

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;
```

```
Record Types
```

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; **Components Rules**

"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;
AdaCore
```

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

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

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

Is the definition legal?



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

Operations

Available Operations

- Predefined
 - Equality (and thus inequality)
 - if A = B then
 - Assignment
 - A := B;
- User-defined
 - Subprograms

Operations

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;

Operations

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

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 \Rightarrow 5.6, Imaginary \Rightarrow 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
```

Aggregates

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 \Rightarrow (16, December, 2001));
```

Aggregates with Only One Component

Must use named form

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

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

```
Record Types
```

Aggregates

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);
AdaCore
```

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

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
 Compilation error
 D. Run-time error
The aggregate is incomplete. The aggregate must specify all
```

components. You could use box notation (A => 1, others => <>)

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
```

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
 Compilation error
 Run-time error
```

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

AdaCore

Aggregates

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
 ■ X := (1, '2', 4, (others => 5))
```

Aggregates

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;
    : constant Nested T := (others => -1);
Ζ
Which assignment(s) is (are) legal?
 A X := (1, '2', Three => 3, Four => (6))
 \blacksquare X := (Two \Rightarrow '2', Four \Rightarrow Z, others \Rightarrow 5)
 \mathbf{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 Y is not initialized
 D Positional for all components
```

Delta Aggregates



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

AdaCore

l Types

Default Values

Default Values

Default Values

Component Default Values

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

Default Component Value Evaluation

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

record

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

```
Record Types
Default Values
```

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

Default Values

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?
```

(1, 2, 3)
(1, 1, 100)
(1, 2, 100)
(100, 101, 102)

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?
    M (1, 2, 3)
    B (1, 1, 100)
```

(1, 2, 100) ○ (100, 101, 102)

Explanations

```
C => 100
Multiple declaration calls Next twice
C correct
C => 100 has no effect on A and B
AdaCore
```

Variant Records

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

Variant Records

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

```
Record Types
```

Variant Records

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

```
AdaCo<u>re</u>
```

constraints do not match

```
Record Types
```

Mutable Variant Record

Type will become <u>mutable</u> if its discriminant has a default value and we instantiate the object without specifying a discriminant

```
type Person_Group is (Student, Faculty);
2
   type Person (Group : Person_Group := Student) is -- default value
3
   record
4
      Age : Positive;
      case Group is
6
          when Student =>
7
             Gpa : Float range 0.0 .. 4.0;
8
          when Faculty =>
9
             Pubs : Positive:
10
      end case:
11
   end record;
12
     Pat : Person: is mutable
     Sam : Person (Faculty); is not mutable

    Declaring an object with an explicit discriminant value (Faculty)

            makes it immutable
```

```
Record Types
```

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 \dots -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 \dots -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?

Variant_Object.F, Variant_Object.Flag
 Variant_Object.F
 None: Compilation error
 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?

Variant_Object.F, Variant_Object.Flag
Variant_Object.F *None: Compilation error*None: Run-time error

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

Record Types			
Lab			

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

Lab

Record Types Lab Solution - Declarations

```
with Ada.Text IO; use Ada.Text IO;
1
   procedure Main is
\mathbf{2}
3
      type Name T is array (1 .. 6) of Character;
4
      type Index_T is range 0 .. 1_000;
5
      type Queue T is array (Index T range 1 .. 1 000) of Name T;
6
7
      type Fifo_Queue_T is record
8
          Next_Available : Index_T := 1;
9
          Last Served : Index T := 0;
10
          Queue : Queue_T := (others => (others => ' '));
11
      end record;
12
13
      Queue : Fifo_Queue_T;
14
      Choice : Integer;
15
         AdaCore
                                                       307 / 928
```

```
Record Types
```

Lab

Record Types Lab Solution - Implementation

begin 18 1000 19 Put ("1 = add to queue | 2 = remove from queue | others => done: "): 20 Choice := Integer'Value (Get Line); if Choice = 1 then 22 Put ("Enter name: "): 23 Queue.Queue (Queue.Next Available) := Name T (Get Line); Queue.Next Available := Queue.Next Available + 1: 25elsif Choice = 2 then if Queue.Next Available = 1 then Put_Line ("Nobody in line"); 28 else Queue.Last Served := Queue.Last Served + 1; Put_Line ("Now serving: " & String (Queue.Queue (Queue.Last_Served))); 31 end if; else exit: 34 end if: New Line; 36 end loop; 37 28 Put Line ("Remaining in line: "); 39 for Index in Queue,Last Served + 1 ... Queue,Next Available - 1 loop 40 Put Line (" " & String (Queue.Queue (Index))); end loop; 42 43 end Main; 44

Summary

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

Su				

Introduction

Introduction

```
Subprograms
```

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;
```

```
Subprograms
```

Introduction

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;
```

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;</pre>
  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;
```

```
Subprograms
```

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

Parameters

```
Subprograms
```

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);
```

Parameters

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

```
Subprograms
Parameters
```

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;
```

Parameters

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

Parameters

By-Copy Vs By-Reference Types

Ву-Сору

- 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

AdaCore 333/928
```

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;

R := Subtract (V2, V1);

```
Subprograms
```

Parameters

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;
```

Parameters

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?
```

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

Parameters

Quiz

```
G 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

AdaCore

Null Procedures

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

```
Subprograms
Null Procedures
```

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

Nested Subprograms

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

1 procedure Main is

```
2
      function Read (Prompt : String) return Types.Line T is
3
      begin
4
          Put (Prompt & "> ");
5
          return Types.Line_T'Value (Get_Line);
6
      end Read;
7
8
      Lines : Types.Lines_T (1 .. 10);
9
   begin
10
      for J in Lines'Range loop
11
          Lines (J) := Read ("Line " & J'Image);
12
      end loop;
13
```

Procedure Specifics

Procedure Specifics

```
Subprograms
```

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
```

Procedure Specifics

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

```
Subprograms
```

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;
```

```
Subprograms
```

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
```

```
Subprograms
```

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;
```

```
Subprograms
```

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;</pre>
```

```
Subprograms
```

Function Dynamic-Size Results

```
function Char Mult (C : Character; L : Natural)
  return String is
   R : String (1 \dots 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

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);

```
Subprograms
```

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?

- Expression functions cannot be nested functions.
- 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?

- Expression functions cannot be nested functions.
- 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

- M They can be nested subprograms (just like any other subprogram)
- As in other subprograms, the implementation can serve as the specification
- G Because they are expressions, the return statement is not allowed
- An expression function does not allow assignment statements, but it can call another function that is **not** an expression function.

Potential Pitfalls

Potential Pitfalls

```
Subprograms
```

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;
```

Potential Pitfalls

"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
 - $X \rightarrow 0, Y \rightarrow 1$ (if Global evaluated first)
 - $X \rightarrow 1, Y \rightarrow 1$ (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" AdaCore 363/928

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

Extended 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

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

Extended Example

Primitive Operations for the Set

- Create a set
 - function Make (Colors : Group_Of_Colors_T) return Set_T;
- Add a color to the set
- 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;

AdaCore

```
Subprograms
```

Extended Example

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));
```

AdaCore

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	:	\texttt{Set}_T	:=	Rgb	or 1	Light;
Intersection	:	\texttt{Set}_T	:=	Rgb	and	Light;
Difference	:	\texttt{Set}_T	:=	Rgb	xor	Light;

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;
4
      function Search
6
        (List : List T;
         Item : Integer)
8
         return Positive is
9
      begin
10
         if List'Length = 0 then
             return 1;
         elsif Item <= List (List'First) then
13
             return 1;
14
         else
             for Idx in (List'First + 1) .. List'Length loop
16
                if Item <= List (Idx) then
                   return Idx:
18
                end if:
19
             end loop;
20
             return List'Last:
21
         end if:
      end Search;
23
           AdaCore
```

Subprograms Lab Solution - Main

```
procedure Add (Item : Integer) is
25
         Place : Natural := Search (List (1..Length), Item);
26
      begin
27
         if List (Place) /= Item then
28
             Length
                                          := Length + 1;
29
             List (Place + 1 .. Length) := List (Place .. Length - 1);
30
            List (Place)
                                        := Item:
         end if;
32
      end Add:
33
34
   begin
35
36
      Add (100):
37
      Add (50);
38
      Add (25):
39
      Add (50):
40
      Add (90);
41
      Add (45):
42
      Add (22);
43
44
      for Idx in 1 .. Length loop
45
         Put_Line (List (Idx)'Image);
46
      end loop;
47
48
   end Main;
49
```

Summary

Summary

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

Type Derivation

Introduction

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

```
Type Derivation
```

Introduction

Reminder: What is a Type?

A type is characterized by two components

- 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 Derivation

```
Type Derivation
```

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 : Noturel T := 0;
```

- $N_Obj : Natural_T := 0;$
 - \blacksquare I_Obj := N_Obj; \rightarrow 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);
```

```
Type Derivation
```

Simple Derivation

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

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;

```
Type Derivation
```

Primitives

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

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

AdaCore

```
Type Derivation
Primitives
```

Creating Primitives for Children

- Just because we can inherit a primitive from our 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);

```
Type Derivation
```

Primitives

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)

Infinite recursion (will result in Storage_Error at run-time)
 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
```

AdaCore

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

Expressions: In-Depth

Introduction

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 Tests

Membership Tests

"Membership" Operation

Syntax

- 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)"

Membership Tests

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
```

Membership Tests

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

- 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
- Legal returns True if Today is Tue or Thu

Qualified Names

Qualified Names

Qualification

- Explicitly indicates the subtype of the value
- Syntax

- Similar to conversion syntax
 - Mnemonic "qualification uses quote"
- Various uses shown in course
 - Testing constraints
 - Removing ambiguity of overloading
 - Enhancing readability via explicitness

```
Expressions: In-Depth
```

Qualified Names

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

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)

Conditional Expressions

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)");
```

Conditional Expressions

Boolean "If Expressions"

- Return a value of either True or False
 - \blacksquare (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

Conditional Expressions

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)

Conditional Expressions

"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

```
. . .
```

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);
```

Conditional Expressions

"Case Expression" Example

```
Leap : constant Boolean :=
   (Today.Year mod 4 = 0 and Today.Year mod 100 \neq 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 \Rightarrow 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);</li>
B F := Sqrt (if X < 0.0 then -1.0 * X else X);</li>
C B := (if X < 0.0 then Sqrt (-1.0 * X) < 10.0 else True);</li>
D B := (if X < 0.0 then Sqrt (-1.0 * X) < 10.0);</li>
```

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);</li>
B F := Sqrt (if X < 0.0 then -1.0 * X else X);</li>
C B := (if X < 0.0 then Sqrt (-1.0 * X) < 10.0 else True);</li>
D B := (if X < 0.0 then Sqrt (-1.0 * X) < 10.0);</li>
```

Explanations

- A. Missing parentheses around expression
- 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 $X \ge 0.0$

Quantified Expressions



Introduction

- Expressions that have a Boolean value
- The value indicates something about a set of objects
 - In particular, whether something is True about that set
- That "something" is expressed as an arbitrary boolean expression
 - A so-called "predicate"
- "Universal" quantified expressions
 - Indicate whether predicate holds for all components
- "Existential" quantified expressions
 - Indicate whether predicate holds for at least one component

Semantics Are As If You Wrote This Code

```
function Universal (Set : Components) return Boolean is
begin
 for C of Set loop
    if not Predicate (C) then
     return False; -- Predicate must be true for all
    end if:
  end loop;
 return True:
end Universal:
function Existential (Set : Components) return Boolean is
begin
 for C of Set loop
    if Predicate (C) then
      return True; -- Predicate need only be true for one
    end if:
  end loop;
 return False:
end Existential;
```

AdaCore

Quantified Expressions Syntax

Four for variants

- Index-based in or component-based of
- Existential some or universal all

■ Using arrow => to indicate *predicate* expression

(for some Index in Subtype_T => Predicate (Index))
(for all Index in Subtype_T => Predicate (Index))
(for some Value of Container_Obj => Predicate (Value))
(for all Value of Container_Obj => Predicate (Value))

Simple Examples

Values : constant array (1 .. 10) of Integer := (...); Is_Any_Even : constant Boolean := (for some V of Values => V mod 2 = 0); Are_All_Even : constant Boolean := (for all V of Values => V mod 2 = 0);

Universal Quantifier

- In logic, denoted by \forall (inverted 'A', for "all")
- "There is no member of the set for which the predicate does not hold"

■ If predicate is False for any member, the whole is False

Functional equivalent

```
function Universal (Set : Components) return Boolean is
begin
for C of Set loop
if not Predicate (C) then
return False; -- Predicate must be true for all
end if;
end loop;
return True;
end Universal;
AdaCore
427/928
```

Universal Quantifier Illustration

- "There is no member of the set for which the predicate does not hold"
- Given a set of integer answers to a quiz, there are no answers that are not 42 (i.e., all are 42)

```
Ultimate_Answer : constant := 42; -- to everything...
Answers : constant array (1 .. 10)
of Integer := (...);
All_Correct_1 : constant Boolean :=
  (for all Component of Answers =>
      Component = Ultimate_Answer);
All_Correct_2 : constant Boolean :=
    (for all K in Answers'Range =>
      Answers (K) = Ultimate_Answer);
```

Universal Quantifier Real-World Example

```
type DMA_Status_Flag is (...);
function Status_Indicated (
   Flag : DMA_Status_Flag)
   return Boolean;
None_Set : constant Boolean := (
   for all Flag in DMA_Status_Flag =>
      not Status_Indicated (Flag));
```

Existential Quantifier

- In logic, denoted by \exists (rotated 'E', for "exists")
- "There is at least one member of the set for which the predicate holds"

■ If predicate is True for any member, the whole is True

Functional equivalent

Existential Quantifier Illustration

- "There is at least one member of the set for which the predicate holds"
- Given set of Integer answers to a quiz, there is at least one answer that is 42

```
Ultimate_Answer : constant := 42; -- to everything...
Answers : constant array (1 .. 10)
of Integer := (...);
Any_Correct_1 : constant Boolean :=
  (for some Component of Answers =>
      Component = Ultimate_Answer);
Any_Correct_2 : constant Boolean :=
    (for some K in Answers'Range =>
      Answers (K) = Ultimate_Answer);
```

Index-Based Vs Component-Based Indexing

Given an array of Integers

Values : constant array (1 .. 10) of Integer := (...);

Component-based indexing is useful for checking individual values

Contains_Negative_Number : constant Boolean :=
 (for some N of Values => N < 0);</pre>

Index-based indexing is useful for comparing across values

```
Is_Sorted : constant Boolean :=
  (for all I in Values'Range =>
    I = Values'First or else
    Values (I) >= Values (I-1));
```

"Pop Quiz" for Quantified Expressions

What will be the value of Ascending_Order? Table : constant array (1 .. 10) of Integer := (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);Ascending_Order : constant Boolean := (for all K in Table'Range => K > Table'First and then Table (K - 1) <= Table (K));Answer: False. Predicate fails when K = Table'First First subcondition is False! Condition should be Ascending_Order : constant Boolean := (for all K in Table'Range => K = Table'First or else Table $(K - 1) \leq Table (K)$;

When the Set Is Empty ...

- Universally quantified expressions are True
 - Definition: there is no member of the set for which the predicate does not hold
 - If the set is empty, there is no such member, so True
 - "All people 12-feet tall will be given free chocolate."
- Existentially quantified expressions are False
 - Definition: there is at least one member of the set for which the predicate holds
- If the set is empty, there is no such member, so False
- Common convention in set theory, arbitrary but settled

Not Just Arrays: Any "Iterable" Objects

- Those that can be iterated over
- Language-defined, such as the containers
- User-defined too

package Characters is new Ada.Containers.Vectors (Positive, Character); use Characters; Alphabet : constant Vector := To_Vector ('A',1) & 'B' & 'C'; Any_Zed : constant Boolean := (for some C of Alphabet => C = 'Z'); All_Lower : constant Boolean := (for all C of Alphabet => Is_Lower (C));

Conditional / Quantified Expression Usage

- Use them when a function would be too heavy
- Don't over-use them!
 - if (for some Component of Answers =>
 Component = Ultimate_Answer)
 then
- Function names enhance readability
 - So put the quantified expression in a function

if At_Least_One_Answered (Answers) then

 Even in pre/postconditions, use functions containing quantified expressions for abstraction

Quantified Expressions

Quiz

Which declaration(s) is (are) legal?

- A. function F (S : String) return Boolean is
 (for all C of S => C /= ' ');
- B. function F (S : String) return Boolean is (not for some C of S => C = ' ');
- C. function F (S : String) return String is (for all C of S => C);
- D function F (S : String) return String is (if (for all C of S => C /= ' ') then "OK" else "NOK");

Quantified Expressions

Quiz

Which declaration(s) is (are) legal?

- A. function F (S : String) return Boolean is
 (for all C of S => C /= ' ');
- B. function F (S : String) return Boolean is
 (not for some C of S => C = ' ');
- C. function F (S : String) return String is (for all C of S => C);
- D function F (S : String) return String is
 (if (for all C of S => C /= ' ') then "OK"
 else "NOK");

B. Parentheses required around the quantified expressionC. Must return a Boolean

AdaCore

Quantified Expressions

Quiz

```
type T1 is array (1 .. 3) of Integer;
type T2 is array (1 .. 3) of Integer;
```

Which piece(s) of code correctly perform(s) equality check on A and B?

- Image function "=" (A : T1; B : T2) return Boolean is
 (A = T1 (B));
- B function "=" (A : T1; B : T2) return Boolean is
 (for all E1 of A => (for all E2 of B => E1 = E2));
- function "=" (A : T1; B : T2) return Boolean is
 (for some E1 of A => (for some E2 of B => E1 =
 E2));
- D function "=" (A : T1; B : T2) return Boolean is (for all J in A'Range => A (J) = B (J));

Quiz

```
type T1 is array (1 .. 3) of Integer;
type T2 is array (1 .. 3) of Integer;
```

Which piece(s) of code correctly perform(s) equality check on A and B?

Image function "=" (A : T1; B : T2) return Boolean is
 (A = T1 (B));

B function "=" (A : T1; B : T2) return Boolean is
 (for all E1 of A => (for all E2 of B => E1 = E2));

```
G function "=" (A : T1; B : T2) return Boolean is
  (for some E1 of A => (for some E2 of B => E1 =
  E2));
```

```
function "=" (A : T1; B : T2) return Boolean is
 (for all J in A'Range => A (J) = B (J));
```

B. Counterexample: A = B = (0, 1, 0) returns False

Counterexample: A = (0, 0, 1) and B = (0, 1, 1) returns True

AdaCore

Quiz

```
type Array1_T is array (1 .. 3) of Integer;
type Array2_T is array (1 .. 3) of Array1_T;
A : Array2_T; -- array of arrays (of 3 components each)
```

Which expression could be used to determine if at least one of $\ensuremath{\mathbb{A}}$'s components are sorted?

```
    M (for some El of A => (for some Idx in 2 .. 3 => El (Idx) >= El (Idx - 1)));
    G (for all El of A => for all Idx in 2 .. 3 => El (Idx) >= El (Idx - 1)));
    G (for some El of A => (for all Idx in 2 .. 3 => El (Idx) >= El (Idx - 1)));
    D (for all El of A => (for some Idx in 2 .. 3 =>
```

El (Idx) >= El (Idx - 1));

Quiz

```
type Array1_T is array (1 .. 3) of Integer;
type Array2 T is array (1 .. 3) of Array1 T;
A : Array2 T; -- array of arrays (of 3 components each)
Which expression could be used to determine if at least one of A's
components are sorted?
 A (for some El of A => (for some Idx in 2 .. 3 =>
      El (Idx) \ge El (Idx - 1)):
 ■ (for all El of A => for all Idx in 2 ... 3 =>
      El (Idx) >= El (Idx - 1)):
 (for some El of A => (for all Idx in 2 .. 3 =>
      El (Idx) >= El (Idx - 1)):
 ▶ (for all El of A => (for some Idx in 2 ... 3 =>
       El (Idx) >= El (Idx - 1));
 M Will be True if any component has two consecutive increasing
    values
 B. Will be True if every component is sorted
 Correct
 D Will be True if every component has two consecutive increasing
```

values

AdaCore

Lab

Expressions Lab

Requirements

- Allow the user to fill a list with dates
- After the list is created, use *quantified expressions* to print True/False
 - If any date is not legal (taking into account leap years!)
 - If all dates are in the same calendar year
- Use expression functions for all validation routines

Hints

- Use subtype membership for range validation
- You will need *conditional expressions* in your functions
- You can use component-based iterations for some checks
 - But you must use indexed-based iterations for others
- This is the same lab as the *Expressions* lab, we're just replacing the validation functions with quantified expressions!
 - So you can just copy that project and update the code!

Lab

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:
         Dav : Positive:
      end record:
      List : array (1 .. 5) of Date_T;
14
      Item : Date T:
      function Is_Leap_Year (Year : Positive)
18
                              return Boolean is
        (Year mod 400 = 0 or else (Year mod 4 = 0 and Year mod 100 \neq 0);
20
      function Days In Month (Month : Positive:
                               Year : Positive)
                               return Dav T is
        (case Month is when 4 \mid 6 \mid 9 \mid 11 \Rightarrow 30,
            when 2 => (if Is Leap Year (Year) then 29 else 28), when others => 31);
      function Is Valid (Date : Date T)
                          return Boolean is
28
        (Date.Year in Year T and then Date.Month in Month T
29
         and then Date.Day <= Days In Month (Date.Month, Date.Year));
30
      function Any Invalid return Boolean is
         (for some Date of List => not Is Valid (Date));
33
35
      function Same Year return Boolean is
         (for all I in List'Range => List (I).Year = List (List'First).Year);
```

Lab

Expressions Lab Solution - Main

```
function Number (Prompt : String)
37
                        return Positive is
38
      begin
39
         Put (Prompt & "> "):
40
          return Positive'Value (Get Line);
41
      end Number;
42
43
   begin
44
45
      for I in List'Range loop
46
          Item.Year := Number ("Year"):
47
         Item.Month := Number ("Month");
48
         Item.Day := Number ("Day");
49
         List (I) := Item:
50
      end loop;
51
52
      Put Line ("Any invalid: " & Boolean'Image (Any Invalid));
53
      Put Line ("Same Year: " & Boolean'Image (Same Year));
54
55
   end Main:
56
```

Summary

Summary

Summary

- Conditional expressions are allowed wherever expressions are allowed, but beware over-use
 - Especially useful when a constant is intended
 - Especially useful when a static expression is required
- Quantified expressions are general purpose but especially useful with pre/postconditions
 - Consider hiding them behind expressive function names

Overloading

0		-1	- 1 H	1.2.2	
U	vei	ric	ad	IIN	g

Introduction

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)
```

```
Overloading
```

Introduction

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

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;
```

Enumerals and Operators

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
```

Enumerals and Operators

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

AdaCore

Call Resolution

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:
```

Call Resolution

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?

```
M P := Horizontal_T'(Middle) * Middle;
B P := Top * Right;
P := "*" (Middle, Top);
P := "*" (H => Middle, V => Top);
```

Call Resolution

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?

M 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
- When overloading subprogram names, best to not just switch the order of parameters

AdaCore

User-Defined Equality

User-Defined Equality

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*

Ove	erloading				
Lal	b				

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

AdaCore

Lab

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 \Rightarrow 4, when Five \Rightarrow 5.
                       when Six => 6, when Seven => 7,
27
                       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' \Rightarrow Four, when 5' \Rightarrow 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)));
38
```

Lab

Overloading Lab Solution - Main

```
Last Entry : Digit T := 0:
   begin
      1000
         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;
76 end Main;
```

Summary

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

AdaCore



Introduction

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

Introduction

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];
```

Introduction

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

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;
```

Referencing Other Packages

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)

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

Referencing Other Packages

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;
```

Bodies

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;
```

Bodies

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;
```

AdaCore

Packages

Bodies

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:
```

Packages

Bodies

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
 Parameter V is out but not assigned (legal but not a good idea)
 Redeclaration of Object One
 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

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;
```

Packages

Executable Parts

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;

Idioms

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;
```

Idioms

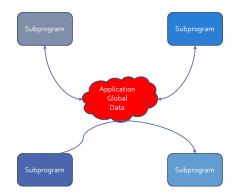
Group of Related Program Units

- 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
 - \blacksquare 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



Packages Idioms





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...

Packages Idioms

Packages	
Lab	

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

AdaCore

508 / 928

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 \rightarrow 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

Lab

2

Packages Lab Solution - Constants

1 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;
```

9 end Constants;

Lab

Packages Lab Solution - Input

```
with Constants;
   package Input is
2
      function Get_Value (Prompt : String) return Constants.Integer_T;
3
   end Input;
4
5
   with Ada.Text_IO; use Ada.Text_IO;
6
   package body Input is
8
      function Get Value (Prompt : String) return Constants.Integer T is
9
         Ret Val : Integer;
10
      begin
         Put (Prompt & "> "):
         1000
13
             Ret_Val := Integer'Value (Get_Line);
             exit when Ret Val >= Constants.Lowest Value
               and then Ret Val <= Constants.Highest Value;
16
             Put ("Invalid. Try Again >");
         end loop;
18
         return Ret_Val;
19
      end Get Value:
20
21
   end Input;
22
```

Packages Lab Solution - List

: package List is procedure Add (Value : Integer); procedure Remove (Value : Integer); function Length return Natural: procedure Print: end List: s with Ada.Text_IO; use Ada.Text_IO; with Constants: 10 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 29 Content (I .. Last - 1) := Content (I + 1 .. Last); 30 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); 45 end List;

Lab

Packages Lab Solution - Main

```
with Ada.Text_IO; use Ada.Text_IO;
   with Input;
   with List:
   procedure Main is
   begin
      100p
         Put ("(A)dd | (R)emove | (P)rint | (Q)uit : "):
9
         declare
10
            Str : constant String := Get_Line;
11
         begin
12
            exit when Str'Length = 0;
            case Str (Str'First) is
               when 'A' =>
                  List.Add (Input.Get_Value ("Value to add"));
16
               when 'R' =>
                  List.Remove (Input.Get Value ("Value to remove"));
18
               when 'P' =>
                  List.Print;
               when 'Q' =>
                  exit;
               when others =>
                  Put Line ("Illegal entry");
            end case;
         end;
      end loop;
28
   end Main:
29
```

Summary

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

Private Types		
Introduction		

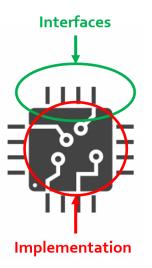
Introduction

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 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

Implementing Abstract Data Types Via Views

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 <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;

Implementing Abstract Data Types Via Views

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 partPhysically 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

Implementing Abstract Data Types Via Views

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;
```

Implementing Abstract Data Types Via Views

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
 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
- Correct type will have a known size at run-time

Private Part Construction

Private Part Construction

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

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;
```

Private Part Construction

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
 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
 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.

AdaCore

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

```
Private Types
```

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

```
Private Types
```

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 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 components is annoying

```
type Vector is array (Positive range <>) of Float;
V : Vector (1 .. 3);
...
V (1) := Alpha; -- Illegal since Vector is private
```

Pri	vate	1 V	nes

Idioms

Idioms

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

```
Private Types
Idioms
```

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:
```

Private Types

Idioms

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);
```

Private Types		
Lab		

Private Types Lab

Requirements

- Implement a program to create a map such that
 - Map key is a description of a flag
 - Map component 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 component
 - 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; 12 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'Inage (First) else ""); begin if First = Last then return Str; 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)); 46 end Colors;

Private Types Lab Solution - Flag Map (Spec)

```
with Colors:
  package Flags is
      type Key T is (USA, England, France, Italy);
      type Map Component T is private;
      type Map T is private;
      procedure Add (Map : in out Map_T;
                     Kev
                                          Kev T:
                     Description :
                                         Colors.Color Set T:
                     Success
                                      out Boolean):
      procedure Remove (Map ; in out Map T;
11
                        Kev
                                         Kev T:
                        Success : out Boolean);
      procedure Modify (Map
                             : in out Map T;
                        Key
                                             Key T;
                        Description :
                                            Colors.Color Set T;
16
                        Success
                                        out Boolean);
18
      function Exists (Map : Map_T; Key : Key_T) return Boolean;
      function Get (Map : Map_T; Key : Key_T) return Map_Component_T;
      function Image (Item : Map_Component_T) return String;
      function Image (Flag ; Map T) return String;
22
   private
23
      type Map Component T is record
24
         Key
                    : Key T := Key T'First;
25
         Description : Colors.Color Set T := Colors.Empty Set;
26
27
      end record:
      type Map Array T is array (1 .. 100) of Map Component T;
28
      type Map T is record
29
         Values : Map Array T:
         Length : Natural := 0;
      end record:
   end Flags;
33
```

Private Types Lab Solution - Flag Map (Body - 1 of 2)

```
function Find (Map : Map_T;
                     Kev : Kev 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;
                     Kev
                                          Kev 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_Component_T :=
                 (Kev
                             -> Kev.
                  Description => Description):
            begin
               Map.Length
                                      := Map.Length + 1;
               Map.Values (Map.Length) := New_Iten;
30
               Success
                                      := True;
            end;
         end if;
      end Add;
      procedure Remove (Map
                               : in out Map_T;
36
                        Key
                                         Key_T;
                        Success : out Boolean) is
         Index : constant Integer := Find (Map, Key);
28
      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_Component_T is
   Index : constant Integer := Find (Map, Key);
   Ret Val : Map Component 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_Component_T) return String is
  (Item.Kev'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
         Iten : constant Map_Component_T := Flag.Values (I);
         Str : constant String
                                        := Inage (Item):
      begin
         Ret Val (Next .. Next + Str'Length) := Image (Item) & ASCII.LF:
         Nort
                                            := 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
      1000
         Put ("Enter country name (");
         for Key in Flags.Key_T loop
            Put (Flags.Kev T'Image (Kev) & " "):
         end loop:
         Put ("): ");
         declare
            Str
                        : constant String := Get Line;
16
            Key
                        : Flags.Key T;
            Description : Colors.Color Set T;
            Success
                        : Boolean;
19
20
         begin
            exit when Str'Length = 0;
            Key
                        := Flags.Key T'Value (Str);
22
            Description := Input.Get;
            if Flags, Exists (Map, Kev) then
               Flags.Modify (Map, Key, Description, Success);
            else
               Flags.Add (Map, Key, Description, Success);
            end if:
         end:
      end loop;
30
31
32
      Put Line (Flags.Image (Map));
   end Main;
33
```

Summary

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
 - More on this later
- Common interface design idioms have arisen
 - Resulting from representation independence
- Assume private types as initial design choice
 - Change is inevitable

Program Structure

Introduction

Introduction

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

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

- 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

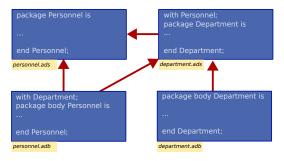
Handling Cyclic Dependencies

- 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 an *incomplete type*

```
    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

Circular Dependencies

Legal Package Declaration Dependency

```
with Department;
package Personnel is
  type Employee is private;
 procedure Assign (This : in Employee;
                     To : in out Department.Section);
private
 type Employee is record
    Assigned To : Department.Section;
  end record;
end Personnel;
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;
```

AdaCore

```
Program Structure
```

Circular Dependencies

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;

AdaCore

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

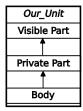
Solution: Hierarchical Library Units

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

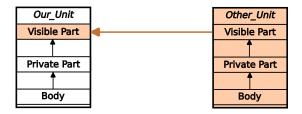
In a **package**, the **body** sees everything the **private part** sees, and the **private part** sees everything the **visible part** sees.



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.

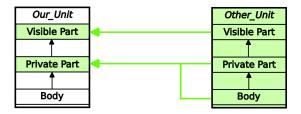
Another **package** can see our **visible part** (depending on where the "with" is), but nothing else.



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.

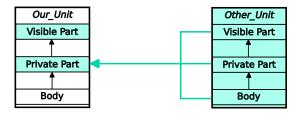
Our child's visible part can see our visible part, and its private part (and body) can see our private part



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;
```

```
AdaCore
```

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;

AdaCo<u>re</u>

. . .

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:
```

AdaCore

Predefined Hierarchies

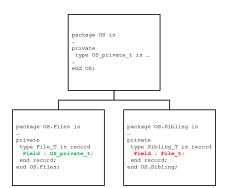
Standard library facilities are children of Ada

- 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?
    Parent.Parent Object + Parent.Sibling.Sibling Object
```

```
B Parent_Object + Sibling_Sibling_Object
G Parent_Object + Sibling_Object
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?

```
    Parent.Parent_Object + Parent.Sibling_Object
    Parent_Object + Sibling_Sibling_Object
    Parent_Object + Sibling_Object
    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.

AdaCore

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

AdaCore

```
Program Structure
```

Correlation to C++ Class Visibility Controls

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;
```

```
Program Structure
```

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
  backing
```

```
procedure Reset is
begin
   Top := 0;
end Reset;
end Stack.Reset;
```

```
Program Structure
```

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?

4.	return	Object_A;	
-			

- B. return Object_B;
 - return Object_C;
- 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;
 - . 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

- 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;
      AdaCore
```

```
Program Structure
```

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;

AdaCore

Private Children

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;

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

AdaCore

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: 26 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): 39 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;
10
   package body Messages.Modify is
13
      procedure Kind (Message : in out Message_T;
                      New Value :
                                         Kind T) is
      begin
         Message.Kind := New Value;
      end Kind:
18
19
      procedure Request (Message : in out Message_T;
20
                         New Value :
                                            Request T) is
      begin
22
         Message.Request := New Value;
23
      end Request;
24
25
      procedure Status (Message : in out Message_T;
                                           Status T) is
                        New Value :
      begin
         Message.Status := New Value;
      end Status:
   end Messages.Modify;
32
```

Program Structure Lab Solution - Main

```
with Ada.Text IO; use Ada.Text IO;
1
   with Messages;
2
   with Messages.Modify;
3
   procedure Main is
4
      Message : Messages.Message_T;
5
      procedure Print is
6
      begin
         Put Line ("Kind => " & Messages.Kind (Message)'Image);
8
         Put_Line ("Request => " & Messages.Request (Message)'Image);
9
         Put_Line ("Status => " & Messages.Status (Message)'Image);
10
         New Line:
      end Print:
   begin
13
      Message := Messages.Create (Kind => Messages.Command.
14
                                    Request \Rightarrow 12.34,
15
                                    Status => 56):
16
      Print:
      Messages.Modify.Request (Message => Message,
18
                                 New Value => 98.76):
19
      Print;
20
   end Main:
21
```

Summary

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

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 Pkg; provides direct visibility into public items in Pkg

- Direct Visibility being used
 - as if object was referenced from within package
- 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 Pkq_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;
       AdaCore
```

No Ambiguity

AdaCore

```
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;
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:
```

AdaCore

"use type" and "use all type" Clauses

"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

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

end Types;

AdaCore

"use" Clauses Comparison

Blue = context clause being used	Red = comp
No "use" clause	
<pre>with Get_Distance; with Types; package Example is no context clause</pre>	with Get_Di: with Types; package Exam 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);	Point0 : 1 Point1 : 1 Point2 : 1 Point3 : 1 Point4 : 1
end Example;	end Example.
"use type" clause	
<pre>with Get_Distance; with Types; package Example is use type Types.Distance;</pre>	with Get_Di: with Types; package Exam use all ty
<pre>Point0 : Distance_T := Get_Distance; Point1 : Types.Distance_T := Get_Distance;</pre>	Point0 : Point1 : 1

Point2 : Types.Distance T := Get Distance; Point3 : Types.Distance T := (Point1 - Point2) / 2; Point4 : Types.Distance T := Min (Point1, Point2);

end Example;

pile errors with the context clause

"use" clause

```
istance;
mple is
5;
```

Distance T := Get Distance; Types.Distance T := Get Distance; Types.Distance T := Get Distance; Types.Distance T := (Point1 - Point2) / 2; Types.Distance T := Min (Point1, Point2);

"use all type" clause

istance; mple is ype Types.Distance;

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" and "use all type" Clauses

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;
```

AdaCore

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 $A^{**2} = B^{**2} + C^{**2} C^{**2}$

```
-- 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)

AdaCore

The "renames" Keyword

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

package Trig renames Math. Trigonometry

Objects (or components of objects)

Subprograms

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;
```

Visibility	
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 components whose length is Number_of_Sides
- Create a main program that will
 - Create an object of each Shape_T
 - Set the values for each component in Shape_T
 - Add all the components in each object and print the total

Hints

There are multiple ways to resolve this!

AdaCore

Visibility Lab Solution - Types

```
package Quads is
1
2
       Number Of Sides : constant Natural := 4;
3
       type Side T is range 0 ... 1 000;
4
       type Shape_T is array (1 .. Number_Of_Sides) of Side_T;
5
6
   end Quads;
7
8
   package Triangles is
9
10
       Number_Of_Sides : constant Natural := 3;
11
       type Side_T is range 0 .. 1_000;
12
       type Shape T is array (1 .. Number Of Sides) of Side T;
13
14
   end Triangles;
15
         AdaCore
                                                        636 / 928
```

Visibility Lab Solution - Main #1

```
with Ada.Text IO; use Ada.Text IO;
   with Quads;
   with Triangles:
   procedure Main1 is
4
      use type Quads.Side T:
6
      Q Sides : Natural renames Quads.Number Of Sides:
              : Quads.Shape_T := (1, 2, 3, 4);
      Quad
      Quad Total : Quads.Side T := 0:
9
      use type Triangles.Side T;
      T Sides : Natural renames Triangles.Number Of Sides:
12
      Triangle : Triangles.Shape T := (1, 2, 3);
13
      Triangle Total : Triangles.Side T := 0;
14
15
16
   begin
17
      for I in 1 .. Q Sides loop
18
         Quad Total := Quad Total + Quad (I);
      end loop;
20
      Put_Line ("Quad: " & Quads.Side_T'Image (Quad_Total));
^{22}
23
      for I in 1 .. T Sides loop
         Triangle Total := Triangle Total + Triangle (I);
24
      end loop;
25
      Put Line ("Triangle: " & Triangles.Side T'Image (Triangle Total));
26
27
   end Main1;
28
```

Visibility Lab Solution - Main #2

```
with Ada.Text IO; use Ada.Text IO;
2 with Quads: use Quads:
   with Triangles; use Triangles;
3
   procedure Main2 is
4
      function Q_Image (S : Quads.Side_T) return String
         renames Quads.Side T'Image:
6
      Quad : Quads.Shape T := (1, 2, 3, 4);
      Quad Total : Quads.Side T := 0;
8
9
      function T Image (S : Triangles, Side T) return String
10
         renames Triangles.Side T'Image;
11
      Triangle : Triangles.Shape_T := (1, 2, 3);
12
      Triangle Total : Triangles.Side T := 0:
13
14
15
   begin
16
17
      for I in Quad'Range loop
         Quad Total := Quad Total + Quad (I);
18
      end loop:
19
      Put Line ("Quad: " & Q Image (Quad Total));
20
^{21}
      for I in Triangle'Range loop
22
         Triangle Total := Triangle Total + Triangle (I):
23
      end loop;
^{24}
      Put Line ("Triangle: " & T_Image (Triangle_Total));
25
26
   end Main2;
27
```

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

Tagged Derivation: An Introduction

Introduction

Introduction

Introduction

Object-Oriented Programming with Tagged Types

For record types

type T is tagged record

- Child types can add new components (*attributes*)
- Object of a child type can be substituted for base type
- Primitive (*method*) can *dispatch* at run-time depending on the type at call-site
- Types can be extended by other packages
 - Conversion and qualification to base type is allowed
- Private data is encapsulated through privacy

Introduction

Tagged Derivation Ada Vs C++

```
type T1 is tagged record
                               class T1 {
  Member1 : Integer;
                                 public:
end record;
                                   int Member1;
                                   virtual void Attr F(void);
procedure Attr_F (This : T1); };
type T2 is new T1 with record class T2 : public T1 {
  Member2 : Integer;
                                 public:
end record;
                                   int Member2;
                                   virtual void Attr_F(void);
overriding procedure Attr_F (
                                   virtual void Attr F2(void)
     This : T2);
                                }:
procedure Attr_F2 (This : T2);
```

Tagged Derivation

Tagged Derivation

```
Tagged Derivation: An Introduction
```

Tagged Derivation

Difference with Simple Derivation

Tagged derivation can change the structure of a type

```
Keywords tagged record and with record
```

```
type Root is tagged record
  F1 : Integer;
end record;
```

```
type Child is new Root with record
F2 : Integer;
end record;
```

```
Root_Object : Root := (F1 => 101);
Child_Object : Child := (F1 => 201, F2 => 202);
```

Type Extension

- A tagged derivation has to be a type extension
 - Use with null record if there are no additional components

type Child is new Root with null record; type Child is new Root; -- illegal

Conversion is only allowed from child to parent

```
V1 : Root;
V2 : Child;
....
V1 := Root (V2);
V2 := Child (V1); -- illegal
```

Information on extending private types appears at the end of this module

Primitives

Child cannot remove a primitive

- Child can add new primitives
- Controlling parameter

Parameters the subprogram is a primitive of
 For tagged types, all should have the same type
 type Root1 is tagged null record;
 type Root2 is tagged null record;

Tagged Derivation: An Introduction

Tagged Derivation

Freeze Point for Tagged Types

Freeze point definition does not change

- A variable of the type is declared
- The type is derived
- The end of the scope is reached

Declaring tagged type primitives past freeze point is forbidden

type Root is tagged null record;

procedure Prim (V : Root);

type Child is new Root with null record; -- freeze root

```
procedure Prim2 (V : Root); -- illegal
```

V : Child; -- freeze child

procedure Prim3 (V : Child); -- illegal

Tagged Derivation: An Introduction

Tagged Derivation

Tagged Aggregate

At initialization, all components (including inherited) must have a value

```
type Root is tagged record
    F1 : Integer;
end record;
```

```
type Child is new Root with record
F2 : Integer;
end record;
```

```
V : Child := (F1 => 0, F2 => 0);
```

■ For **private types** use *aggregate extension*

```
    Copy of a parent instance
```

Use with null record absent new components

```
V2 : Child := (Parent_Instance with F2 => 0);
V3 : Empty_Child := (Parent_Instance with null record);
```

Information on aggregates of private extensions appears at the end of this module

AdaCore

Tagged Derivation

Overriding Indicators

Optional overriding and not overriding indicators

```
type Shape_T is tagged record
Name : String (1..10);
end record;
```

```
-- primitives of "Shape_T"
function Get_Name (S : Shape_T) return String;
procedure Set_Name (S : in out Shape_T);
```

```
-- Derive "Point_T" from Shape_T
type Point_T is new Shape_T with record
Origin : Coord_T;
end record;
```

```
-- We want to _change_ the behavior of Set_Name
overriding procedure Set_Name (P : in out Point_T);
-- We want to _add_ a new primitive
not overriding procedure Set_Origin (P : in out Point_T);
-- We get "Get_Name" for free
```

AdaCore

Prefix Notation

- Tagged types primitives can be called as usual
- The call can use prefixed notation
 - If the first argument is a controlling parameter
 - No need for use or use type for visibility

-- Prim1 visible even without *use Pkg* X.Prim1;

```
declare
    use Pkg;
begin
    Prim1 (X);
end;
```

Tagged Derivation

Quiz

```
Which declaration(s) will make P a primitive of T1?
 A type T1 is tagged null record;
   procedure P (O : T1) is null;
 B type TO is tagged null record;
   type T1 is new T0 with null record;
   type T2 is new T0 with null record;
   procedure P (0 : T1) is null:
 C. type T1 is tagged null record;
   Object : T1;
   procedure P (O : T1) is null;
 D package Nested is
     type T1 is tagged null record;
    end Nested:
   use Nested:
   procedure P (O : T1) is null;
```

Tagged Derivation

Quiz

```
Which declaration(s) will make P a primitive of T1?
 A type T1 is tagged null record;
    procedure P (O : T1) is null;
 B. type TO is tagged null record;
    type T1 is new T0 with null record;
    type T2 is new T0 with null record:
    procedure P (O : T1) is null:
 C. type T1 is tagged null record;
    Object : T1;
    procedure P (O : T1) is null;
 D package Nested is
      type T1 is tagged null record;
    end Nested:
    use Nested:
    procedure P (0 : T1) is null;
 A Primitive (same scope)
 B. Primitive (T1 is not yet frozen)
 C T1 is frozen by the object declaration
 D Primitive must be declared in same scope as type
```

Quiz

with Shapes; -- Defines tagged type Shape, with primitive P
with Colors; use Colors; -- Defines tagged type Color, with primitive P
with Weights; -- Defines tagged type Weight, with primitive P
use type Weights.Weight;

```
procedure Main is
   The_Shape : Shapes.Shape;
   The_Color : Colors.Color;
   The_Weight : Weights.Weight;
```

Which statement(s) is (are) valid?

```
A. The_Shape.P
B. P (The_Shape)
C. P (The_Color)
D. P (The Weight)
```

Quiz

with Shapes; -- Defines tagged type Shape, with primitive P
with Colors; use Colors; -- Defines tagged type Color, with primitive P
with Weights; -- Defines tagged type Weight, with primitive P
use type Weights.Weight;

procedure Main is
 The_Shape : Shapes.Shape;
 The_Color : Colors.Color;
 The_Weight : Weights.Weight;

Which statement(s) is (are) valid?

```
A The_Shape.P
B P (The_Shape)
C P (The_Color)
D P (The_Weight)
D use type only gives y
```

use type only gives visibility to operators; needs to be use all type

AdaCore

Tagged Derivation

Quiz

Which code block(s) is (are) legal?

A. type A1 is record Component1 : Integer; end record: type A2 is new A1 with null record;



```
B. type B1 is tagged record
      Component2 : Integer;
    end record:
    type B2 is new B1 with record
      Component2b : Integer;
    end record:
```

C. type C1 is tagged record Component3 : Integer; end record: type C2 is new C1 with record Component3 : Integer; end record:

D. type D1 is tagged record Component1 : Integer; end record; type D2 is new D1;

Tagged Derivation

Quiz

Which code block(s) is (are) legal?



```
B. type B1 is tagged record
Component2 : Integer;
end record;
type B2 is new B1 with r.
```

```
type B2 is new B1 with record
Component2b : Integer;
end record;
```

Explanations

- C. type C1 is tagged record Component3 : Integer; end record; type C2 is new C1 with record Component3 : Integer; end record;
- D. type D1 is tagged record Component1 : Integer; end record; type D2 is new D1;
- A. Cannot extend a non-tagged type
- B. Correct
- C. Components must have distinct names
- D. Types derived from a tagged type must have an extension

Lab

Lab

Tagged Derivation Lab

- Requirements
 - Create a type structure that could be used in a business
 - A person has some defining characteristics
 - An employee is a *person* with some employment information
 - A staff member is an *employee* with specific job information
 - Create primitive operations to read and print the objects
 - Create a main program to test the objects and operations
- Hints
 - Use overriding and not overriding as appropriate (Ada 2005 and above)

Tagged Derivation Lab Solution - Types (Spec)

: package Employee is subtype Name_T is String (1 .. 6); type Date_T is record Year : Positive; Month : Positive: Day : Positive; end record: type Job_T is (Sales, Engineer, Bookkeeping); type Person_T is tagged record The Name : Name T: The_Birth_Date : Date_T; end record: procedure Set_Name (0 : in out Person_T; Value : Name T): function Name (0 : Person_T) return Name_T; procedure Set Birth Date (0 : in out Person T: 20 Value : Date T): function Birth_Date (0 : Person_T) return Date_T; procedure Print (0 : Person T): -- Employee -type Employee_T is new Person_T with record The Employee Id : Positive: The Start Date : Date T: end record; not overriding procedure Set_Start_Date (0 : in out Employee_T; Value : Date_T); not overriding function Start_Date (0 : Employee_T) return Date_T; overriding procedure Print (0 : Employee_T); -- Position -type Position_T is new Employee_T with record The Job : Job T: end record; not overriding procedure Set Job (0 : in out Position T: Value : Job_T); not overriding function Job (0 : Position T) return Job T: overriding procedure Print (0 : Position_T);

end Employee;

AdaCore

Tagged Derivation Lab Solution - Types (Partial Body)

```
1 with Ada.Text IO: use Ada.Text IO:
   package body Employee is
      function Image (Date : Date T) return String is
        (Date, Year'Image & " - " & Date, Month'Image & " - " & Date, Dav'Image);
      procedure Set Name (0 : in out Person T;
                          Value :
                                         Name T) is
      begin
         O.The Name := Value;
      end Set Name;
      function Name (0 : Person T) return Name T is (0. The Name):
      procedure Set Birth Date (0 : in out Person T;
                                Value :
                                              Date T) is
      begin
         0. The Birth Date := Value:
      end Set Birth Date;
      function Birth Date (0 : Person T) return Date T is (0. The Birth Date);
      procedure Print (0 : Person T) is
      begin
         Put Line ("Name: " & O.Name);
         Put Line ("Birthdate: " & Image (0.Birth Date)):
      end Print:
      not overriding procedure Set Start Date
        (0 : in out Employee T:
         Value :
                        Date T) is
      begin
         O.The Start Date := Value;
      end Set Start Date:
      not overriding function Start Date (0 : Employee T) return Date T is
         (0.The Start Date);
      overriding procedure Print (0 : Employee T) is
      begin
         Put Line ("Name: " & Name (0));
38
         Put Line ("Birthdate: " & Image (0.Birth Date));
         Put Line ("Startdate: " & Image (0.Start Date)):
      end Print:
42
```

Lab

Tagged Derivation Lab Solution - Main

```
with Ada.Text IO; use Ada.Text IO;
   with Employee;
   procedure Main is
      Applicant : Employee.Person T;
              : Employee.Employee T;
      Employ
      Staff
                : Employee.Position T:
   begin
       Applicant.Set Name ("Wilma ");
      Applicant.Set Birth Date ((Year => 1 234.
10
                                  Month => 12.
                                  Day => 1));
      Employ.Set Name ("Betty ");
14
      Employ.Set Birth Date ((Year => 2 345,
15
                               Month \Rightarrow 11.
                               Dav => 2));
      Employ.Set Start Date ((Year => 3 456,
18
                               Month \Rightarrow 10.
19
                               Dav => 3));
20
21
      Staff.Set Name ("Bambam");
22
      Staff.Set Birth Date ((Year => 4 567.
                              Month => 9.
24
                              Day => 4));
25
      Staff.Set Start Date ((Year => 5 678,
26
                              Month => 8.
                              Day => 5));
28
      Staff.Set Job (Employee.Engineer);
29
30
      Applicant.Print;
31
      Employ.Print;
33
       Staff.Print:
34 end Main:
```

AdaCore

Summary

Summary

Summary

Summary

- Tagged derivation
 - Building block for OOP types in Ada
- Primitives rules for tagged types are trickier
 - Primitives forbidden below freeze point
 - Unique controlling parameter
 - Tip: Keep the number of tagged type per package low

Extending Tagged Types

Extending Tagged Types

Tagged Derivation: An Introduction

Extending Tagged Types

How Do You Extend a Tagged Type?

```
Premise of a tagged type is to extend an existing type
In general, that means we want to add more components
    We can extend a tagged type by adding components
  package Animals is
    type Animal_T is tagged record
      Age : Natural;
    end record;
  end Animals:
  with Animals: use Animals:
  package Mammals is
    type Mammal T is new Animal T with record
      Number Of Legs : Natural;
    end record:
  end Mammals:
  with Mammals; use Mammals;
  package Canines is
    type Canine_T is new Mammal_T with record
      Domesticated : Boolean:
    end record:
  end Canines;
```

Tagged Aggregate

- At initialization, all components (including inherited) must have a value
 - Animal : Animal T := $(Age \Rightarrow 1);$ Mammal : Mammal T := (Age => 2. Number_Of_Legs => 2); Canine : Canine T := (Age => 2. Number_Of_Legs \Rightarrow 4. Domesticated => True):

But we can also "seed" the aggregate with a parent object

Mammal := (Animal with Number Of Legs => 4); Canine := (Animal with Number Of Legs => 4, Domesticated => False); Canine := (Mammal with Domesticated => True); AdaCore

Extending Tagged Types

Private Tagged Types

- But data hiding says types should be private!
- So we can define our base type as private

```
package Animals is
type Animal_T is tagged private;
function Get_Age (P : Animal_T) return Natural;
procedure Set_Age (P : in out Animal_T; A : Natural);
private
type Animal_T is tagged record
Age : Natural;
end Animals;
```

And still allow derivation

```
with Animals;
package Mammals is
type Mammal_T is new Animals.Animal_T with record
Number_Of_Legs : Natural;
end record;
```

But now the only way to get access to Age is with accessor subprograms

AdaCore

Private Extensions

- In the previous slide, we exposed the components for Mammal_T!
- Better would be to make the extension itself private

```
package Mammals is
  type Mammal_T is new Animals.Animal_T with private;
  private
   type Mammal_T is new Animals.Animal_T with record
      Number_Of_Legs : Natural;
  end record;
end Mammals;
```

Aggregates with Private Tagged Types

- Remember, an aggregate must specify values for all components
 - But with private types, we can't see all the components!
- So we need to use the "seed" method:

```
procedure Inside_Mammals_Pkg is
Animal : Animal_T := Animals.Create;
Mammal : Mammal_T;
begin
Mammal := (Animal with Number_Of_Legs => 4);
Mammal := (Animals.Create with Number_Of_Legs => 4);
end Inside_Mammals_Pkg;
```

Note that we cannot use others => <> for components that are not visible to us

Null Extensions

To create a new type with no additional components

We still need to "extend" the record - we just do it with an empty record

type Dog_T is new Canine_T with null record;

• We still need to specify the "added" components in an aggregate

C : Canine_T := Canines.Create; Dog1 : Dog_T := C; -- Compile Error Dog2 : Dog_T := (C with null record);

Quiz

```
Given the following code:
package Parents is
  type Parent_T is tagged private;
  function Create return Parent T:
private
  type Parent_T is tagged record
     Id : Integer;
  end record;
end Parents;
with Parents; use Parents;
package Children is
  P : Parent T;
  type Child T is new Parent T with record
     Count : Natural;
  end record;
  function Create (C : Natural) return Child T:
end Children:
Which completion(s) of Create is (are) valid?
 M function Create return Child_T is (Parents.Create
   with Count => 0);
 B function Create return Child_T is (others => <>);
 function Create return Child T is (0, 0):
 function Create return Child T is (P with Count =>
   0);
```

Quiz

```
Given the following code:
package Parents is
  type Parent_T is tagged private;
  function Create return Parent T:
private
  type Parent_T is tagged record
     Id : Integer;
  end record;
end Parents:
with Parents; use Parents;
package Children is
  P : Parent T;
  type Child T is new Parent T with record
     Count : Natural;
  end record:
  function Create (C : Natural) return Child T:
end Children:
```

Which completion(s) of Create is (are) valid?

```
    function Create return Child_T is (Parents.Create

    with Count => 0);

    function Create return Child_T is (others => <>);

    function Create return Child_T is (0, 0);

    function Create return Child_T is (P with Count =>

    0);
```

Explanations

- Correct Parents.Create returns Parent_T
- Cannot use others to complete private part of an aggregate
- Aggregate has no visibility to Id component, so cannot assign
- D. Correct P is a Parent_T

Exceptions

Introduction

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;
AdaCore 673/928
```

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;
```

```
Exceptions
```

Introduction

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;
```

Exceptions			
Handlers			

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;
    AdaCore
```

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; 683 / 928

Handling Specific Statements¹ 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;
     AdaCore
```

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;
```

Ex			

Handlers

Quiz

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

What will get printed? A. One, Two, Three B. Two, Three C. Two D. Three

Handlers

Quiz

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

What will get printed? Α. One, Two, Three Β. Two, Three Two Three D Explanations A. One is never printed, as although (A - C) is not in the range of 0 ... 100, this is only checked on assignment (so after the addition of B). B. Correct С. 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 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

User-Defined Exceptions

Syntax

```
<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

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

Propagation Demo

1	procedure Do_Something is	16	
2	Error : exception;	17	
3	procedure Unhandled is	18	
4	begin	19	
5	Maybe_Raise (1);	20	
6	<pre>end Unhandled;</pre>	21	
7	procedure Handled is	22	
8	begin		
9	Unhandled;		
10	Maybe_Raise (2);		
11	exception		
12	when Error =>		
13	Print ("Handle 1 or	2")	;
14	end Handled;		
	AdaCore		

6	<pre>begin Do_Something</pre>
7	Maybe_Raise (3);
8	Handled;
9	exception
0	when Error =>
1	<pre>Print ("Handle 3");</pre>
2	<pre>end Do_Something;</pre>

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;
```

Exceptions

Propagation

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;
- s 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
- when Constraint_Error => Put_Line ("Constraint Error");
- when Program_Error => Put_Line ("Program Error");
- 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

Exceptions

Propagation

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;
- s elsif P = 0 then
- 9 raise Main_Problem;
- 10 end if;
- 11 end F;
- 12 begin
- I := F(Input Value);
- Put Line ("Success"):
- 15 exception
- when Constraint_Error => Put_Line ("Constraint Error");
- when Program_Error => Put_Line ("Program Error");
- when others => Put_Line ("Unknown problem");

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

- A. Unknown Problem
- B Success
- Constraint Error
- D Program Error

Explanations

- "Unknown Problem" is printed by the when others due to the raise on line 9 when P is 0
- \blacksquare "Success" is printed when 0 < P < Integer'Last
- Trying to add 1 to P on line 7 generates a Constraint_Error
- \blacksquare 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 <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

Raise Expressions

Expression raising specified exception at run-time

Exceptions			
Lab			

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

Exceptions Lab Solution - Numeric Types

```
1 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;
10
   package body Numeric Types is
      function Legal (C : Character) return Boolean is
13
      begin
         return
           C in '0' .. '9' or C = '+' or C = '-' or C = ' ' or C = 'e' or C = 'E';
      end Legal;
18
      function Value (Str : String) return Integer_T is
19
      begin
20
         for I in Str'Range loop
            if not Legal (Str (I)) then
               raise Illegal_String;
            end if:
25
         end loop:
         return Integer_T'Value (Str);
      exception
         when Constraint Error =>
            raise Out Of Range;
      end Value:
32 end Numeric_Types;
```

Lab

Exceptions Lab Solution - Main

```
with Ada.Text IO:
   with Numeric Types:
   procedure Main is
      procedure Print_Value (Str : String) is
5
         Value : Numeric Types.Integer T:
      begin
         Ada.Text IO.Put (Str & " => "):
8
         Value := Numeric Types.Value (Str);
9
         Ada.Text IO.Put Line (Numeric Types.Integer T'Image (Value));
10
      exception
11
         when Numeric Types.Out Of Range =>
12
            Ada.Text IO.Put Line ("Out of range");
         when Numeric Types.Illegal String =>
14
            Ada.Text IO.Put Line ("Illegal entry");
15
      end Print Value;
16
   begin
18
      Print Value ("123"):
19
      Print_Value ("2_3_4");
20
      Print Value ("-345"):
21
      Print Value ("+456");
22
      Print Value ("1234567890"):
23
      Print Value ("123abc"):
24
      Print Value ("12e3"):
25
   end Main:
26
```

Summary

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



```
Exceptions
```

Summary

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;
```

AdaCore

Summary

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

Access Types

Access Types			

Introduction

Access Types Design

- A memory-addressed object is called an *access type*
- Objects are associated to pools of memory
 - With different allocation / deallocation policies
- Access objects are guaranteed to always be meaningful
 - In the absence of Unchecked_Deallocation
 - And if pool-specific

```
Ada
```

```
type Integer_Pool_Access
is access Integer;
```

```
P_A : Integer_Pool_Access
```

```
:= new Integer;
```

```
type Integer_General_Access
```

- is access all Integer;
- G : aliased Integer;
- G_A : Integer_General_Access := G'Access;

```
■ C++
```

```
int * P_C = malloc (sizeof (int));
int * P_CPP = new int;
```

```
int * G_C = &Some_Int;
```

Access Types - General vs Pool-Specific

General Access Types

- Point to any object of designated type
- Useful for creating aliases to existing objects
- Point to existing object via
 'Access or created by new
- No automatic memory management

Pool-Specific Access Types

- Tightly coupled to dynamically allocated objects
- Used with Ada's controlled memory management (pools)
- Can only point to object created by new
- Memory management tied to specific storage pool

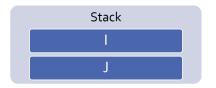
Access Types Can Be Dangerous

- Multiple memory issues
 - Leaks / corruptions
- Introduces potential random failures complicated to analyze
- Increase the complexity of the data structures
- May decrease the performances of the application
 - Dereferences are slightly more expensive than direct access
 - Allocations are a lot more expensive than stacking objects
- Ada avoids using accesses as much as possible
 - Arrays are not pointers
 - Parameters are implicitly passed by reference
- Only use them when needed

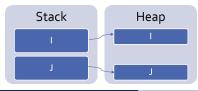
Introduction

Stack Vs Heap

- I : Integer := 0;
- J : String := "Some Long String";



- I : Access_Int := new Integer'(0);
- J : Access_Str := new String'("Some Long String");



AdaCore

Access Types		
Access Types		

Access Types

Declaration Location

Can be at library level

```
package P is
  type String_Access is access String;
end P;
```

Can be nested in a procedure

```
package body P is
    procedure Proc is
        type String_Access is access String;
    begin
        ...
    end Proc;
end P;
```

- Nesting adds non-trivial issues
 - Creates a nested pool with a nested accessibility
 - Don't do that unless you know what you are doing! (see later)

AdaCore

Null Values

- A pointer that does not point to any actual data has a null value
- Access types have a default value of null
- null can be used in assignments and comparisons

```
declare
  type Acc is access all Integer;
  V : Acc;
begin
  if V = null then
      -- will go here
  end if;
  V := new Integer'(0);
  V := null; -- semantically correct, but memory leak
```

Access Types and Primitives

- Subprogram using an access type are primitive of the access type
 - Not the type of the accessed object

type A_T is access all T; procedure Proc (V : A_T); -- Primitive of A_T, not T

- Primitive of the type can be created with the access mode
 - Anonymous access type
 - Details elsewhere

procedure Proc (V : access T); -- Primitive of T

Dereferencing Access Types

- .all does the access dereference
 - Lets you access the object pointed to by the pointer
- .all is optional for
 - Access on a component of an array
 - Access on a component of a record

Access Types

Dereference Examples

```
type R is record
 F1, F2 : Integer;
end record;
type A_Int is access Integer;
type A_String is access all String;
type A_R is access R;
V_Int := A_Int := new Integer;
V_String : A_String := new String'("abc");
V R : A R := new R;
V Int.all := 0;
V String.all := "cde";
V_String (1) := 'z'; -- similar to V_String.all (1) := 'z';
V R.all := (0, 0);
V R.F1 := 1; -- similar to V R.all.F1 := 1;
```

Pool-Specific Access Types

Pool-Specific Access Types

Pool-Specific Access Types

Pool-Specific Access Type

An access type is a type

```
type T is [...]
type T_Access is access T;
V : T_Access := new T;
```

Conversion is **not** possible between pool-specific access types

Allocations

- Objects are created with the new reserved word
- The created object must be constrained
 - The constraint is given during the allocation
 - V : String_Access := new String (1 .. 10);
- The object can be created by copying an existing object using a qualifier
 - V : String_Access := new String'("This is a String");

Deallocations

- Deallocations are unsafe
 - Multiple deallocations problems
 - Memory corruptions
 - Access to deallocated objects
- As soon as you use them, you lose the safety of your access
- But sometimes, you have to do what you have to do ...
 - There's no simple way of doing it
 - Ada provides Ada.Unchecked_Deallocation
 - Has to be instantiated (it's a generic)
 - Must work on an object, reset to null afterwards

Pool-Specific Access Types

Deallocation Example

```
-- generic used to deallocate memory
with Ada. Unchecked Deallocation;
procedure P is
   type An Access is access A Type;
   -- create instances of deallocation function
   -- (object type, access type)
   procedure Free is new Ada.Unchecked_Deallocation
     (A_Type, An_Access);
   V : An_Access := new A_Type;
begin
   Free (V);
   -- V is now null
end P;
```

General Access Types

General Access Types

Can point to any pool (including stack)

```
type T is [...]
type T_Access is access all T;
V : T_Access := new T;
```

- Still distinct type
- Conversions are possible

type T_Access_2 is access all T; V2 : T_Access_2 := T_Access_2 (V); -- legal

Referencing the Stack

- By default, stack-allocated objects cannot be referenced and can even be optimized into a register by the compiler
- aliased declares an object to be referenceable through an access value
 - V : aliased Integer;
- 'Access attribute gives a reference to the object
 - A : Int_Access := V'Access;
 - 'Unchecked_Access does it without checks

Aliased Objects Examples

```
type Acc is access all Integer;
V, G : Acc;
I : aliased Integer;
V := I'Access:
V.all := 5; -- Same a I := 5
procedure P1 is
  I : aliased Integer;
begin
  G := I'Unchecked Access;
   P2:
   -- Necessary to avoid corruption
   -- Watch out for any of G's copies!
   G := null;
end P1;
procedure P2 is
begin
  G.all := 5;
end P2;
```

```
Access Types
```

Aliased Parameters

 To ensure a subprogram parameter always has a valid memory address, define it as aliased

Ensures 'Access and 'Address are valid for the parameter

procedure Example (Param : aliased Integer);

```
Object1 : aliased Integer;
Object2 : Integer;
-- This is OK
Example (Object1);
```

```
-- Compile error: Object2 could be optimized away
-- or stored in a register
Example (Object2);
```

-- Compile error: No address available for parameter Example (123);

AdaCo<u>re</u>

Quiz

```
type General_T is access all Integer;
type Pool_T is access Integer;
```

Aliased_Object : aliased Integer; Random_Object : Integer;

General_Ptr : General_T; Pool_Specific_Ptr : Pool_T;

Which assignment(s) is (are) legal?

- A. General_Ptr := Random_Object'Access;
- B. General_Ptr := Aliased_Object'Access;
- C Pool_Specific_Ptr := Random_Object'Access;
- Pool_Specific_Ptr := Aliased_Object'Access;

Quiz

```
type General_T is access all Integer;
type Pool_T is access Integer;
```

Aliased_Object : aliased Integer; Random_Object : Integer;

General_Ptr : General_T; Pool_Specific_Ptr : Pool_T;

Which assignment(s) is (are) legal?

- A. General_Ptr := Random_Object'Access;
- B. General_Ptr := Aliased_Object'Access;
- C Pool_Specific_Ptr := Random_Object'Access;
- Pool_Specific_Ptr := Aliased_Object'Access;

'Access is only allowed for general access types (General_T). To use 'Access on an object, the object **must** be aliased.

Accessibility Checks

Accessibility Checks

```
Access Types
```

Accessibility Checks

Introduction to Accessibility Checks (1/2)

The depth of an object depends on its nesting within declarative scopes

```
package body P is
-- Library level, depth 0
00 : aliased Integer;
procedure Proc is
-- Library level subprogram, depth 1
type Acc1 is access all Integer;
procedure Nested is
-- Nested subprogram, enclosing + 1, here 2
02 : aliased Integer:
```

- Objects can be referenced by access types that are at same depth or deeper
 - An access scope must be ≤ the object scope
- type Acc1 (depth 1) can access 00 (depth 0) but not O2 (depth
 2)
- The compiler checks it statically
 - Removing checks is a workaround!
- Note: Subprogram library units are at depth 1 and not 0

Accessibility Checks

Introduction to Accessibility Checks (2/2)

```
Issues with nesting
package body P is
   type T0 is access all Integer;
   AO : TO:
   V0 : aliased Integer;
   procedure Proc is
      type T1 is access all Integer;
      A1 : T1:
      V1 : aliased Integer;
   begin
      A0 := V0'Access:
      -- AO := V1'Access; -- illegal
      A0 := V1'Unchecked Access;
      A1 := V0'Access:
      A1 := V1'Access;
      A1 := T1 (A0);
      A1 := new Integer:
      -- A0 := T0 (A1); -- illegal
  end Proc:
end P:
```

To avoid having to face these issues, avoid nested access types

```
Access Types
```

Dynamic Accessibility Checks

Following the same rules

- Performed dynamically by the runtime
- Lots of possible cases
 - New compiler versions may detect more cases
 - Using access always requires proper debugging and reviewing

```
type Acc is access all Integer;
      0 : Acc:
      Outer : aliased Integer := 123;
      procedure Set_Value (V : access Integer) is
      begin
         Put Line (V.all'Image);
         0 := Acc (V);
      end Set Value;
10
   begin
      Set_Value (Outer'Access);
12
      declare
13
         Inner : aliased Integer := 987;
14
      begin
15
         Set Value (Inner'Access);
16
      end:
17
```

raised PROGRAM_ERROR : main.adb:9 accessibility check failed

AdaCore

741 / 928

```
Access Types
```

Getting Around Accessibility Checks

- Sometimes it is OK to use unsafe accesses to data
- 'Unchecked_Access allows access to a variable of an incompatible accessibility level
- Beware of potential problems!

```
type Acc is access all Integer;
G : Acc;
procedure P is
  V : aliased Integer;
begin
  G := V'Unchecked_Access;
  ...
  Do_Something (G.all);
  G := null; -- This is "reasonable"
end P;
```

AdaCore

Using Access Types for Recursive Structures

- It is not possible to declare recursive structure
- But there can be an access to the enclosing type

```
type Cell; -- partial declaration
type Cell_Access is access all Cell;
type Cell is record -- full declaration
    Next : Cell_Access;
    Some_Value : Integer;
end record;
```

Quiz

```
type Global_Access_T is access all Integer;
Global_Access : Global_Access_T;
Global_Object : aliased Integer;
procedure Proc_Access is
   type Local_Access_T is access all Integer;
   Local_Access : Local_Access_T;
   Local_Object : aliased Integer;
begin
```

Which assignment(s) is (are) legal?

M Global_Access := Global_Object'Access;
B Global_Access := Local_Object'Access;
C Local_Access := Global_Object'Access;
D Local_Access := Local_Object'Access;

Quiz

```
type Global_Access_T is access all Integer;
Global_Access : Global_Access_T;
Global_Object : aliased Integer;
procedure Proc_Access is
   type Local_Access_T is access all Integer;
   Local_Access : Local_Access_T;
   Local_Object : aliased Integer;
begin
```

Which assignment(s) is (are) legal?

Α.	Global_Access	:=	Global_Object'Access;
Β.	Global_Access	:=	<pre>Local_Object'Access;</pre>
C.	Local_Access	:=	Global_Object'Access;
D.	Local_Access	:=	Local_Object'Access;

Explanations

- A. Access type has same depth as object
- B. Access type is not allowed to have higher level than accessed object
- C Access type has lower depth than accessed object
- D. Access type has same depth as object

Memory Corruption

```
Access Types
```

Common Memory Problems (1/3)

Uninitialized pointers

```
declare
  type An_Access is access all Integer;
  V : An_Access;
begin
  V.all := 5; -- constraint error
```

Double deallocation

```
declare
  type An_Access is access all Integer;
  procedure Free is new
     Ada.Unchecked_Deallocation (Integer, An_Access);
  V1 : An_Access := new Integer;
  V2 : An_Access := V1;
begin
  Free (V1);
     ...
  Free (V2);
  May raise Storage_Error if memory is still protected
     (unallocated)
```

- May deallocate a different object if memory has been reallocated
 - Putting that object in an inconsistent state

AdaCore

```
Access Types
```

Common Memory Problems (2/3)

Accessing deallocated memory

```
declare
  type An_Access is access all Integer;
  procedure Free is new
       Ada.Unchecked_Deallocation (Integer, An_Access);
  V1 : An_Access := new Integer;
  V2 : An_Access := V1;
begin
  Free (V1);
  ...
  V2.all := 5;
```

- May raise Storage_Error if memory is still protected (unallocated)
- May modify a different object if memory has been reallocated (putting that object in an inconsistent state)

AdaCore

Common Memory Problems (3/3)

Memory leaks

```
declare
  type An_Access is access all Integer;
  procedure Free is new
      Ada.Unchecked_Deallocation (Integer, An_Access);
  V : An_Access := new Integer;
begin
  V := null;
```

- Silent problem
 - Might raise Storage_Error if too many leaks
 - Might slow down the program if too many page faults

How to Fix Memory Problems?

- There is no language-defined solution
- Use the debugger!
- Use additional tools
 - gnatmem monitor memory leaks
 - valgrind monitor all the dynamic memory
 - GNAT.Debug_Pools gives a pool for an access type, raising explicit exception in case of invalid access
 - Others...

Anonymous Access Types

Anonymous Access Types

Anonymous Access Types

Anonymous Access Parameters

- Parameter modes are of 4 types: in, out, in out, access
- The access mode is called *anonymous access type*
 - Anonymous access is implicitly general (no need for all)
- When used:
 - Any named access can be passed as parameter
 - Any anonymous access can be passed as parameter

```
type Acc is access all Integer;
Aliased_Integer : aliased Integer;
Access_Object : Acc := Aliased_Integer'Access;
procedure P1 (Anon_Access : access Integer) is null;
procedure P2 (Access_Parameter : access Integer) is
begin
```

- P1 (Aliased_Integer'Access);
- P1 (Access_Object);

```
P1 (Access_Parameter);
```

end P2;

Anonymous Access Types

Anonymous Access Types

Other places can declare an anonymous access

```
function F return access Integer;
V : access Integer;
type T (V : access Integer) is record
C : access Integer;
end record;
type A is array (Integer range <>) of access Integer;
```

Do not use them without a clear understanding of accessibility check rules

Anonymous Access Constants

 constant (instead of all) denotes an access type through which the referenced object cannot be modified

```
type CAcc is access constant Integer;
G1 : aliased Integer;
G2 : aliased constant Integer := 123;
V1 : CAcc := G1'Access;
V2 : CAcc := G2'Access;
V1.all := 0; -- illegal
```

- not null denotes an access type for which null value cannot be accepted
 - Available in Ada 2005 and later

```
type NAcc is not null access Integer;
V : NAcc := null; -- illegal
```

Also works for subprogram parameters

```
procedure Bar (V1 : access constant Integer);
procedure Foo (V1 : not null access Integer); -- Ada 2005
```

AdaCo<u>re</u>

Access Types			
Lab			

Lab

Access Types Lab

Overview

- Create a (really simple) Password Manager
 - The Password Manager should store the password and a counter for each of some number of logins
 - As it's a Password Manager, you want to modify the data directly (not pass the information around)

Requirements

- Create a Password Manager package
 - Create a record to store the password string and the counter
 - Create an array of these records indexed by the login identification
 - The user should be able to retrieve a pointer to the record, either for modification or for viewing
- Main program should:
 - Set passwords and initial counter values for many logins
 - Print password and counter value for each login

Hint

- Password is a string of varying length
 - Easiest way to do this is a pointer to a string that gets initialized to the correct length

AdaCore

Lab

Access Types Lab Solution - Password Manager

```
package Password Manager is
   type Login T is (Email, Banking, Amazon, Streaming);
   type Password T is record
      Count
              : Natural:
      Password : access String:
   end record:
   type Modifiable T is access all Password T:
   type Viewable T is access constant Password T:
   function Update (Login : Login T) return Modifiable T:
   function View (Login : Login T) return Viewable T;
end Password Manager:
package body Password Manager is
   Passwords : array (Login T) of aliased Password T:
   function Update (Login : Login T) return Modifiable T is
      (Passwords (Login) 'Access);
   function View (Login : Login T) return Viewable T is
      (Passwords (Login) 'Access);
```

end Password_Manager;

AdaCore

Lab

Access Types Lab Solution - Main

```
with Ada.Text IO: use Ada.Text IO:
   with Password Manager; use Password Manager;
2
   procedure Main is
3
4
      procedure Update (Which : Password_Manager.Login_T;
5
                         Pw
                               : String;
                         Count : Natural) is
      begin
8
         Update (Which).Password := new String'(Pw);
9
         Update (Which).Count := Count:
      end Update:
11
   begin
13
      Update (Email, "QWE!@#", 1);
14
      Update (Banking, "asd123", 22);
      Update (Amazon, "098poi", 333);
16
      Update (Streaming, ")(*LKJ", 444);
18
      for Login in Login_T'Range loop
19
         Put Line
20
           (Login'Image & " => " & View (Login).Password.all &
21
            View (Login).Count'Image):
      end loop:
23
   end Main;
^{24}
```

Summary

Summary

Summary

- Access types are the same as C/C++ pointers
- There are usually better ways of memory management
 - Language has its own ways of dealing with large objects passed as parameters
 - Language has libraries dedicated to memory allocation / deallocation
- At a minimum, create your own generics to do allocation / deallocation
 - Minimize memory leakage and corruption

Genericity

Introduction

The Notion of a Pattern

```
Sometimes algorithms can be abstracted from types and
 subprograms
  procedure Swap_Int (Left, Right : in out Integer) is
    V : Integer := 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;
```

AdaCo<u>re</u>

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);
AdaCore
```

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 Data

```
Genericity
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
<pre>type T7 (<>) is private;</pre>	(<>) 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);
```

AdaCore

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 Component (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 Component is new Component
   (T => String,
```

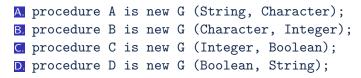
```
Acc => String_Ptr,
Index => Integer,
Arr => String_Array);
```

Generic Data

Quiz

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

Which is (are) legal instantiation(s)?



Generic Data

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:
 - \blacksquare in \rightarrow read only
 - in out \rightarrow read write
- Generic variables can be defined after generic types

```
Generic package
generic
type Component_T is private;
Array_Size : Positive;
High_Watermark : in out Component_T;
package Repository is
Generic instance
V : Positive := 10;
Max : Float;
procedure My_Repository is new Repository
(Component_T => Float,
Array_size => V,
High Watermark => Max);
```

```
Genericity
```

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

AdaCore

Quiz

```
generic
  type Component_T is (<>);
  Last : in out Component_T;
  procedure Write (P : Component_T);
Numeric : Integer;
Enumerated : Boolean;
Floating_Point : Float;
Which of the following piece(s) of code is (are) legal?
M procedure Write_A is new Write (Integer, Numeric)
procedure Write_B is new Write (Boolean, Enumerated)
procedure Write_C is new Write (Integer, Integer'Pos
 (Numeric))
```

```
procedure Write_D is new Write (Float,
Floating_Point)
```

Quiz

```
generic
  type Component_T is (<>);
  Last : in out Component_T;
procedure Write (P : Component_T);
```

```
Numeric : Integer;
Enumerated : Boolean;
Floating_Point : Float;
```

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

```
procedure Write_A is new Write (Integer, Numeric)
procedure Write_B is new Write (Boolean, Enumerated)
procedure Write_C is new Write (Integer, Integer'Pos (Numeric))
procedure Write_D is new Write (Float, Floating_Point)
Legal
Legal
The second generic parameter has to be a variable
```

```
D. The first generic parameter has to be discrete
```

Genericity

Generic Formal Data

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?

```
function IncrA is new Incr (Integer, Weird);
function IncrB is new Incr (Record_T, Add);
function IncrC is new Incr (Record_T);
function IncrD is new Incr (Integer);
```

Genericity

Generic Formal Data

Quiz

Given the following generic function:

generic

```
type Some_T is private; with function "+" (L : Some_T; R : Integer) return Some_T is \diamondsuit; 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?
```

```
Inction IncrA is new Incr (Integer, Weird);
If function IncrB is new Incr (Record_T, Add);
If function IncrC is new Incr (Record_T);
If 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.
```

- Weird matches the subprogram profile, so Incr will use Weird when doing addition for Integer
- Add matches the subprogram profile, so Incr will use Add when doing the addition for Record_T
- There is no matching + operation for Record_T, so that instantiation fails to compile
- 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

Generic Completion

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;
```

```
Genericity
```

Generic Completion

Generic Incomplete Parameters

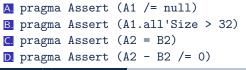
- A generic type can be incomplete
- Allows generic instantiations before full type definition
- Restricts the possible usages (only access)

```
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;
```

Genericity

Quiz

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



Genericity

Quiz

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		

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 components
- Print routine will need to know how to print one component

AdaCore

Genericity Lab Solution - Generic (Spec)

```
generic
      type Component T is private;
      Max Size : Natural:
      with function ">" (L, R : Component T) return Boolean is <>;
4
      with function Image (Component : Component_T) return String;
   package Generic_List is
6
      type List T is private;
8
9
      procedure Add (This : in out List T;
10
                      Item : in Component T):
11
      procedure Sort (This : in out List_T);
12
      procedure Print (List : List T);
13
14
   private
15
      subtype Index T is Natural range 0 .. Max Size;
16
      type List Array T is array (1 .. Index T'Last) of Component T:
17
18
      type List T is record
19
         Values : List_Array_T;
20
         Length : Index T := 0;
21
      end record:
22
   end Generic_List;
23
```

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;
                     Ttem : in
                                   Component T) is
      begin
         This.Length
                                   := This.Length + 1;
         This.Values (This.Length) := Item;
      end Add:
10
      procedure Sort (This : in out List T) is
         Temp : Component_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):
18
                  This.Values (J + 1) := Temp:
               end if:
            end loop;
         end loop;
      end Sort:
25
      procedure Print (List : List_T) is
      begin
26
         for I in 1 .. List.Length loop
            Put Line (Integer'Image (I) & ") " & Image (List.Values (I)));
         end loop;
      end Print:
32 end Generic_List;
```

Genericity Lab Solution - Main

```
with Data Type:
   with Generic List:
   procedure Main is
      package List is new Generic List (Component T => Data Type.Record T,
                                        Max Size => 20.
                                        151
                                                  => Data Type.">".
                                        Image => Data_Type.Image);
      My List : List.List T;
      Component : Data Type.Record T;
10
12
   begin
      List.Add (My_List, (Integer_Component => 111,
13
                          Character Component => 'a'));
14
      List.Add (My List, (Integer Component => 111,
                          Character Component => 'z')):
16
      List.Add (My_List, (Integer_Component => 111,
                          Character_Component => 'A'));
18
      List.Add (My List, (Integer Component => 999,
19
                          Character Component => 'B'));
20
      List.Add (My List, (Integer Component => 999,
                          Character Component => 'Y'));
      List.Add (My_List, (Integer_Component => 999,
23
                          Character_Component => 'b'));
24
      List.Add (My List, (Integer Component => 112,
25
                          Character Component => 'a'));
26
      List.Add (My List, (Integer Component => 998.
                          Character Component => 'z'));
28
29
      List.Sort (My List);
30
      List.Print (My List);
32 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

Tasking

Introduction

Introduction

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

AdaCore

Tasking			
Tasks			

Tasks

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;

AdaCore

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

AdaCore 800/928
```

Accept Statement vs Block

Assume the client can rendezvous with a task with the following entry points:

```
accept Acknowledge;
Put_Line ("acknowledge");
```

```
accept Wait_Until_Completion (S : String) do
   Put_Line ("receive " & S);
end Receive_Message;
```

- When Acknowledge is called ...
 - Task immediately releases the caller
 - ... then continues on to the Put_Line statement
- When Wait_Until_Completion is called ...

Task performs everything between do and end of the block
... then releases the caller

AdaCore

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

Protected Objects

- Multitask-safe accessors to get and set state
- No direct state manipulation
- No concurrent modifications
- limited types (No copies allowed)

Protected Objects

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;

AdaCore

Tasking			
Delays			

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 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

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; -- exit loop
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 exit the loop. (If the exit wasn't there, the loop would restart.)

AdaCore

Some Advanced Concepts

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

AdaCore

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

Tasking			
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
2
      protected Monitor is
3
         procedure Set (Id : Task_Type.Task_Id_T);
4
         function Get return Task_Type.Task_Id_T;
      private
6
          Value : Task Type.Task Id T;
      end Monitor:
8
   end Protected Object;
9
10
   package body Protected Object is
11
      protected body Monitor is
12
          procedure Set (Id : Task Type.Task Id T) is
         begin
14
             Value := Id;
         end Set;
16
         function Get return Task_Type.Task_Id_T is (Value);
17
      end Monitor:
18
   end Protected_Object;
19
```

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;
9
   with Protected_Object;
   package body Task Type is
      task body Task_T is
12
13
          Wait Time : Duration:
         Id
                    : Task Id T;
      begin
         accept Start_Task (Task_Id
                                            : Task Id T;
16
                             Delay_Duration : Duration) do
             Wait Time := Delay Duration;
             Τd
                       := Task Id;
          end Start Task:
20
         loop
^{21}
             select
22
                accept Stop Task;
                exit:
24
             or
                delay Wait Time;
26
               Protected_Object.Monitor.Set (Id);
             end select;
28
         end loop;
       end Task T;
30
   end Task_Type;
31
```

Tasking Lab Solution - Main

```
with Ada.Text IO; use Ada.Text IO;
2 with Protected_Object;
3 with Task_Type;
4 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;
6
      use type Task Type.Task Id T;
   begin
8
9
      T1.Start_Task (1_111, 0.3);
10
      T2.Start Task (2 222, 0.5);
11
      T3.Start Task (3 333, 0.7);
12
13
      for Count in 1 .. 20 loop
14
         This Id := Protected Object.Monitor.Get;
15
         if Last Id /= This Id then
16
            Last Id := This Id;
            Put_Line (Count'Image & "> " & Last_Id'image);
18
         end if:
19
         delay 0.2;
20
      end loop;
21
22
      T1.Stop Task:
23
      T2.Stop Task;
24
      T3.Stop_Task;
25
26
27 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

Subprogram Contracts



Introduction

Introduction

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

Introduction



- Boolean expression expected to be True
- Said to hold when True
- Language-defined pragma

- Raises language-defined Assertion_Error exception if expression does not hold
- The Ada.Assertions.Assert subprogram wraps it

```
package Ada.Assertions is
  Assertion_Error : exception;
  procedure Assert (Check : in Boolean);
  procedure Assert (Check : in Boolean; Message : in String);
end Ada.Assertions;
```

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

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

- 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!

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

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

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 run-time errors

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 run-time errors

Explanations

- A. Principles are sane, contracts extend those
- **B.** See previous slide example
- C. e.g. generic contracts are resolved at compile-time
- A failing contract will cause a run-time error, only extensive (dynamic / static) analysis of contracted code may provide confidence in the absence of runtime errors (AoRTE)

Preconditions and Postconditions

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

 Preconditions and Postconditions

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)
```

Preconditions and Postconditions

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

Defensive Programming

```
Should be replaced by subprogram contracts when possible
procedure Push (The_Stack : Stack) is
Entry_Length : constant Positive := Length (The_Stack);
begin
pragma Assert (not Is_Full (The_Stack)); -- entry condition
[...]
pragma Assert (Length (The_Stack) = 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 (Resource : Peripheral) is
begin
  if not Available (Resource) then
```

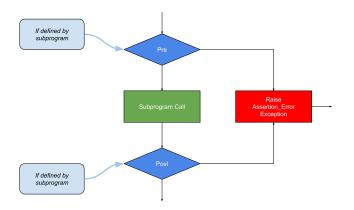
```
-- Corrective action
Force_Release (Resource);
pragma Assert (Available (Resource));
end if;
```

```
Acquire (Resource);
```

end;

Pre/Postcondition Semantics

Calls inserted automatically by compiler



Contract with Quantified Expression

```
Pre- and post-conditions can be arbitrary Boolean expressions
type Status_Flag is (Power, Locked, Running);
```

```
procedure Clear_All_Status (
    Unit : in out Controller)
    -- guarantees no flags remain set after call
    with Post => (for all Flag in Status_Flag =>
        not Status_Indicated (Unit, Flag));
```

```
function Status_Indicated (
    Unit : Controller;
    Flag : Status_Flag)
    return Boolean;
```

Visibility for Subprogram Contracts

- Any visible name
 - All of the subprogram's parameters
 - Can refer to functions not yet specified
 - Must be declared in same scope
 - Different elaboration rules for expression functions

function Top (This : Stack) return Content
with Pre => not Empty (This);
function Empty (This : Stack) return Boolean;

- Post has access to special attributes
 - See later

Preconditions and Postconditions

Preconditions and Postconditions Example

Multiple aspects separated by commas

(Sub)Types Allow Simpler Contracts

Pre-condition

Subtype

```
with
```

```
-- Convert string to Integer
function To_Integer ( S : String ) return Integer
with Pre => S'Length > 0;
procedure Print_Something is
I : Integer := To_Integer ("");
begin
Put_Line (I'Image);
end Print_Something;
Assuming To_Integer is defined somewhere, what happens when
```

Print_Something is run?

"0" is printed
Constraint Error exception
Assertion Error exception
Undefined behavior

```
-- Convert string to Integer
function To_Integer ( S : String ) return Integer
with Pre => S'Length > 0;
procedure Print_Something is
I : Integer := To_Integer ("");
begin
Put_Line (I'Image);
end Print_Something;
Assuming To_Integer is defined somewhere, what happens when
```

Print_Something is run?

"0" is printed
Constraint Error exception
Assertion Error exception
Undefined behavior

Explanations

The call to To_Integer will fail its precondition, which is considered an Assertion_Error exception.

```
function Area (Length : Positive; Height : Positive) return Positive is
  (Length * Height)
with Pre => ?
```

Which pre-condition is necessary for Area to calculate the correct result for all values L and H $\,$

Length > 0 and Height > 0
Length < Positive'Last and Height < Positive'Last
Length * Height in Positive
None of the above

```
function Area (Length : Positive; Height : Positive) return Positive is
  (Length * Height)
with Pre => ?
```

Which pre-condition is necessary for Area to calculate the correct result for all values L and H $\,$

```
    Length > 0 and Height > 0
    Length < Positive'Last and Height < Positive'Last</li>
    Length * Height in Positive
    None of the above
```

Explanations

```
    A Parameters are Positive, so this is unnecessary
    Length = Positive'Last-1 and Height = Positive'Last-1 will still cause an overflow
    Classic trap: the check itself may cause an overflow!
    Preventing an overflow requires using the expression
    Integer'Last / Length <= Height</li>
```

Special Attributes

```
Subprogram Contracts
```

Evaluate an Expression on Subprogram Entry

 Post-conditions may require knowledge of a subprogram's entry context

procedure Increment (This : in out Integer)
with Post => ??? -- how to assert incrementation of `This`?

- Language-defined attribute 'Old
- Expression is evaluated at subprogram entry
 - After pre-conditions check
 - Makes a copy
 - limited types are forbidden
 - May be expensive
 - Expression can be arbitrary
 - Typically in out parameters and globals

```
procedure Increment (This : in out Integer) with
    Pre => This < Integer'Last,
    Post => This = This'Old + 1;
```

Example for Attribute '01d

```
Global : String := Init_Global;
...
-- In Global, move character at Index to the left one position,
-- and then increment the Index
procedure Shift_And_Advance (Index : in out Integer) is
begin
Global (Index) := Global (Index + 1);
Index := Index + 1;
end Shift_And_Advance;
```

Note the different uses of 'Old in the postcondition

```
procedure Shift_And_Advance (Index : in out Integer) with Post =>
   -- Global (Index) before call (so Global and Index are original)
Global (Index)'Old
   -- Original Global and Original Index
   = Global'Old (Index'Old)
   and
   -- Global after call and Index befor call
Global (Index'Old)
        -- Global and Index after call
   = Global (Index);
```

Error on Conditional Evaluation of 'Old

This code is incorrect

Copies In_String (At_Position) on entry

- Will raise an exception on entry if At_Position not in In_String'Range
- The postcondition's if check is not sufficient
- Solution requires a full copy of In_String

Postcondition Usage of Function Results

function result can be read with 'Result

function Greatest_Common_Denominator (A, B : Positive)
 return Positive with

Post => Is_GCD (A, B,

Greatest_Common_Denominator'Result);

Quiz

```
Database : String (1 .. 10) := "ABCDEFGHIJ";

Index : Integer := 4;

-- Set the value for the component at position Index in

-- array Database to Value and then increment Index by 1

function Set_And_Move (Value : Character;

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 ('X', Index)?

Database'Old (Index)

Database (Index'Old)

Quiz

```
Database : String (1 .. 10) := "ABCDEFGHIJ";

Index : Integer := 4;

-- Set the value for the component at position Index in

-- array Database to Value and then increment Index by 1

function Set_And_Move (Value : Character;

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 ('X', Index)?

```
Legend
Value on call entry
Value on call return
```

```
Database'Old (Index)
```

```
Database'Old (Index)
Database before the call: ABCDEFGHIJ
Index after the call : 5
Value : E
```

Database (Index'Old)

Quiz

```
Database : String (1 .. 10) := "ABCDEFGHIJ";

Index : Integer := 4;

-- Set the value for the component at position Index in

-- array Database to Value and then increment Index by 1

function Set_And_Move (Value : Character;

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 ('X', Index)?

```
Legend
Value on call entry
Value on call return
```

```
Database'Old (Index)
```

```
Database'Old (Index)
Database before the call: ABCDEFGHIJ
Index after the call : 5
Value : E
```

```
Database (Index'Old)
```

```
Database (Index'Old)
Database after the call : ABCXEFGHIJ
Index before the call : 4
Value : X
```

Quiz

```
Database : String (1 .. 10) := "ABCDEFGHIJ";

Index : Integer := 4;

-- Set the value for the component at position Index in

-- array Database to Value and then increment Index by 1

function Set_And_Move (Value : Character;

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 ('X', Index)?

```
Legend
Value on call entry
Value on call return
```

```
Database'Old (Index)
```

```
Database'Old (Index)
Database before the call: ABCDEFGHIJ
Index after the call : 5
Value : E
```

```
Database (Index'Old)
```

```
Database (Index'Old)
Database after the call : ABCXEFGHIJ
Index before the call : 4
Value : X
```

```
Database (Index)'Old
Database before the call: ABCDEFGHIJ
Index before the call : 4
Value : D
```

Stack Example (Spec with Contracts)

```
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 and Item = Top'Old;
  function Pop return Integer with
     Pre => not Empty,
     Post => not Full and Pop'Result = Top'Old;
  function Top return Integer with
     Pre => not Empty;
     function Empty return Boolean;
     function Full return Boolean;
     function Full return Boolean;
     function Full return Boolean;
     end Stack_Pkg;
```

package body Stack_Pkg is Values : array (1 .. 100) of Integer: Current : Natural := 0: -- Preconditions prevent Push/Pop failure 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;

In Practice

Pre/Postconditions: to Be or Not to Be

- Preconditions are reasonable default for run-time checks
- Postconditions advantages can be comparatively low
 - Use of 'Old and 'Result with (maybe deep) copy
 - Very useful in static analysis contexts (Hoare triplets)
- For trusted library, enabling preconditions only makes sense
 - Catch user's errors
 - Library is trusted, so Post => True is a reasonable expectation
- Typically contracts are used for validation
- Enabling subprogram contracts in production may be a valid trade-off depending on...
 - Exception failure trace availability in production
 - Overall timing constraints of the final application
 - Consequences of violations propagation
 - Time and space cost of the contracts
- Typically production settings favor telemetry and off-line analysis

```
Subprogram Contracts
```

No Secret Precondition Requirements

- Client should be able to guarantee them
- Enforced by the compiler

```
package Some_Package is
  function Foo return Bar
   with Pre => Hidden; -- illegal private reference
private
  function Hidden return Boolean;
end Some_Package;
```

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));
```

Postcondition Compared to Their Body

Specifying relevant properties may "repeat" the body

- Unlike preconditions
- Typically simpler than the body
- Closer to a re-phrasing than a tautology

Good fit for hard to solve and easy to check problems

- Solvers: Solve (Find_Root'Result, Equation) = 0
- Search: Can_Exit (Path_To_Exit'Result, Maze)
- Cryptography: Match (Signer (Sign_Certificate'Result), Key.Public_Part)
- Bad fit for poorly-defined or self-defining subprograms

function Get_Magic_Number return Integer

with Post => Get_Magic_Number'Result = 42

-- Useless post-condition, simply repeating the body is (42);

Postcondition Compared to Their Body: Example

```
function Greatest Common Denominator (Num1, Num2 : Natural)
  return Integer with
  Post => Is_GCD (Num1,
                   Num2.
                   Greatest Common Denominator'Result);
function Is_GCD (Num1, Num2, Candidate : Integer)
    return Boolean is
  (Num1 rem Candidate = 0 and
   Num2 rem Candidate = 0 and
   (for all K in 1 .. Integer'Min (Num1,Num2) =>
      (if (Num1 rem K = 0 and Num2 rem K = 0)
       then K <= Candidate)));
```

```
Subprogram Contracts
```

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

A function may be unavoidable

Referencing private type components

Subprogram Contracts on Private Types

```
package Bank is
  type Account is private;
  procedure Process_Transaction (This : Account) with
    Pre => This.Balance > 0; -- not legal
  . . .
  function Current Balance (This : Account) return Integer;
  . . .
  procedure R (This : Account) with
    Pre => Current Balance (This) > 0; -- legal
  . . .
private
  type Account is record
    Balance : Natural;
  end record:
  function Current Balance (This : Account) return Integer is
      (This.Balance):
end Bank:
```

```
Subprogram Contracts
```

Preconditions or Explicit Checks?

Any requirement from the spec should be a pre-condition

- If clients need to know the body, abstraction is broken
- With pre-conditions

With defensive code, comments, and return values

```
-- returns True iff push is successful
function Try_Push (This : in out Stack;
Value : Content) return Boolean
begin
```

```
if Full (This) then
return False;
end if;
```

• • •

- But not both
 - For the implementation, preconditions are a guarantee
 - A subprogram body should never test them

Raising Specific Exceptions

- In the Exceptions module, we show how user-defined exceptions are better than pre-defined
 - Stack Push raising Overflow_Error rather than Constraint_Error
- Default behavior for a preconditon failure is Assertion_Error
 - But it doesn't have to be!
- Use *raise expression* in a precondition to get a different exception

Note: Postcondition failure only ever makes sense as an Assertion_Error

It's the supplier's fault, not the client's

Assertion Policy

```
Pre/postconditions 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_r
m/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;
```

Subprogram Contracts Lab

Overview

- Create a priority-based queue ADT
 - Higher priority items come off queue first
 - When priorities are same, process entries in order received

Requirements

- Main program should verify pre-condition failure(s)
 - At least one pre-condition should raise something other than assertion error
- Post-condition should ensure queue is correctly ordered
- Hints
 - Basically a stack, except insertion doesn't necessarily happen at "top"
 - To enable assertions in the runtime from GNAT STUDIO
 - **Edit** \rightarrow Project Properties
 - $\blacksquare \text{ Build} \rightarrow \text{ Switches} \rightarrow \text{ Ada}$
 - Click on Enable assertions

Subprogram Contracts Lab Solution - Queue (Spec)

package Priority Queue is Overflow : exception; type Priority T is (Low, Medium, High); type Queue T is tagged private; subtype String T is String (1 .. 20); procedure Push (Queue : in out Queue T; Priority : Priority T: Value String) with Pre => (not Full (Queue) and then Value'Length > 0) or else raise Overflow, Post => Valid (Queue): procedure Pop (Queue : in out Queue T; Value : out String T) with Pre => not Empty (Queue), Post => Valid (Queue); function Full (Queue : Queue T) return Boolean: function Empty (Queue : Queue T) return Boolean: function Valid (Queue : Queue T) return Boolean; 18 private 19 Max Queue Size : constant := 10; type Entries T is record Priority : Priority T: Value : String T; end record; type Size T is range 0 ... Max Queue Size: type Queue Array T is array (1 .. Size T'Last) of Entries T; type Queue T is tagged record Size : Size_T := 0; Entries : Queue Array T; end record: function Full (Queue : Queue T) return Boolean is (Queue.Size = Size T'Last); function Empty (Queue : Queue T) return Boolean is (Queue.Size = 0); function Valid (Queue : Queue T) return Boolean is (if Queue.Size <= 1 then True else (for all Index in 2 .. Queue.Size => 39 Queue.Entries (Index).Priority >= Queue.Entries (Index - 1).Priority)); 41 end Priority Queue:

Subprogram Contracts Lab Solution - Queue (Body)

package body Priority_Queue is function Pad (Str : String) return String_T is Retval : String T := (others => ' '); begin if Str'Length > Retval'Length then Retval := Str (Str'First .. Str'First + Retval'Length - 1); Retval (1 .. Str'Length) := Str: end if; return Retval: end Pad; procedure Push (Queue : in out Queue T: Priority : Priority_T; Value : String) is Last : Size_T renames Queue.Size; New Entry : constant Entries T := (Priority, Pad (Value)); begin if Queue.Size = 0 then Queue.Entries (Last + 1) := New_Entry; elsif Priority < Queue.Entries (1).Priority then Queue.Entries (2 .. Last + 1) := Queue.Entries (1 .. Last): Queue.Entries (1) := New_Entry; elsif Priority > Queue, Entries (Last), Priority then Queue.Entries (Last + 1) := New_Entry; else for Index in 1 .. Last loop if Priority <= Queue.Entries (Index).Priority then Queue.Entries (Index + 1 .. Last + 1) := Queue.Entries (Index .. Last); Queue.Entries (Index) := New Entry: exit; end if: end loop; end if: Last := Last + 1: end Push: procedure Pop (Queue : in out Queue_T; Value : out String T) is begin := Queue.Entries (Queue.Size).Value: Value Queue.Size := Queue.Size - 1; end Pop; er end Priority_Queue;

Subprograms Contracts Lab Solution - Main

```
with Ada.Text IO; use Ada.Text IO;
   with Priority_Queue;
   procedure Main is
3
      Queue : Priority Queue.Queue T;
      Value : Priority Queue.String T;
   begin
6
      Ada.Text IO.Put Line ("Normal processing");
      for Count in 1 ... 3 loop
0
         for Priority in Priority Queue, Priority T'Range loop
             Queue.Push (Priority, Priority'Image & Count'Image);
         end loop:
      end loop:
      while not Queue.Empty loop
15
         Queue, Pop (Value):
         Put Line (Value);
      end loop;
18
19
      Ada.Text IO.Put Line ("Test overflow");
20
      for Count in 1 .. 4 loop
21
         for Priority in Priority Queue, Priority T'Range loop
22
             Queue.Push (Priority, Priority'Image & Count'Image);
23
          end loop:
^{24}
      end loop;
25
26
   end Main;
27
```

Summary

Summary

Summary

Contract-Based Programming Benefits

- Facilitates building software with reliability built-in
 - Software cannot work well unless "well" is carefully defined
 - Clarifies design by defining requirements/guarantees
- 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 requirements
 - Static analyzers (e.g., SPARK, GNAT Static Analysis Suite) can verify explicit precondition and postconditions

Summary

- Based on viewing source code as clients and suppliers with enforced requirements and guarantees
- No run-time penalties unless enforced
- OOP introduces the tricky issues
 - Inheritance of preconditions and postconditions, for example
- Note that pre/postconditions can be used on concurrency constructs too

	Clients	Suppliers
Preconditions	Requirement	Guarantee
Postconditions	Guarantee	Requirement

Type Contracts



Introduction

Introduction

Strong Typing

We know 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;
```

- But 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

- Add more complicated constraints to a type
- Always enforced, just like other constraints

AdaCore

Type Invariants

Type Invariants

Type Invariants

- There may be conditions that must hold over entire lifetime of objects
 - Pre/postconditions apply only to subprogram calls
- Sometimes low-level facilities can express it

subtype Weekdays is Days range Mon .. Fri;

-- Guaranteed (absent unchecked conversion) Workday : Weekdays := Mon;

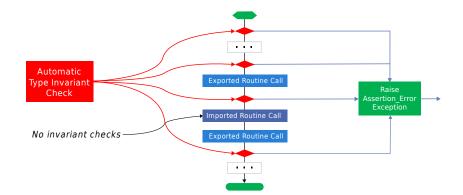
- Type invariants apply across entire lifetime for complex abstract data types
- Part of ADT concept, so only for private types

Type Invariant Verifications

- Automatically inserted by compiler
- Evaluated as postcondition of creation, evaluation, or return object
 - When objects first created
 - Assignment by clients
 - Type conversions
 - Creates new instances
- Not evaluated on internal state changes
 - Internal routine calls
 - Internal assignments
- Remember these are abstract data types

Type Invariants

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;
```

Type Invariants

Example Type Invariant Implementation

package body Bank is . . . function Total (This : Transaction_Vector) return Currency is Result : Currency := 0.0; begin for Value of This loop Result := Result + Value: end loop; return Result: end Total: function Consistent_Balance (This : Account) return Boolean is begin return Total (This.Deposits) - Total (This.Withdrawals) = This.Current Balance; end Consistent_Balance; end Bank:

AdaCore

```
Type Contracts
Type Invariants
```

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;
```

Default Type Initialization for Invariants

- Invariant must hold for initial value
- May need default type initialization to satisfy requirement

```
package Operations is
```

```
-- Type is private, so we can't use Default_Value here
type Private_T is private with Type_Invariant => Zero (Private_T);
procedure Op (This : in out Private_T);
function Zero (This : Private_T) return Boolean;
```

private

```
-- Type is not a record, so we need to use aspect

-- (A record could use default values for its components)

type Private_T is new Integer with Default_Value => 0;

function Zero (This : Private_T) return Boolean is

begin

meture (This = 0);
```

```
return (This = 0);
```

```
end Zero;
```

end Operations;

```
Type Contracts
Type Invariants
```

Type Invariant Clause Placement

Can move aspect clause to private part

```
package Operations is
  type Private_T is private;
  procedure Op (This : in out Private_T);
private
  type Private_T is new Integer with
    Type_Invariant => Private_T = 0,
    Default_Value => 0;
end Operations;
```

It is really an implementation aspect

Client shouldn't care!

Invariants Are Not Foolproof

- Access to ADT representation via pointer could allow back door manipulation
- These are private types, so access to internals must be granted by the private type's code
- Granting internal representation access for an ADT is a highly questionable design!

Type Invariants

Quiz

```
package Counter_Package is
   type Counter T is private:
   procedure Increment (Val : in out Counter_T);
private
   function Check Threshold (Value : Integer)
                                 return Boolean:
   type Counter_T is new Integer with
      Type Invariant => Check Threshold
                        (Integer (Counter T)):
end Counter_Package;
package body Counter Package is
   function Increment_Helper (Helper_Val : Counter_T)
                                return Counter T is
      Next Value : Counter T := Helper Val + 1:
   begin
      return Next_Value;
   end Increment Helper:
   procedure Increment (Val : in out Counter T) is
   begin
      Val := Val + 1;
      Val := Increment Helper (Val);
   end Increment;
   function Check_Threshold (Value : Integer)
                                     return Boolean is
      (Value <= 100); -- check against constraint
end Counter_Package;
```

If **Increment** is called from outside of Counter_Package, how many times is **Check_Threshold** called?

A. 1

B. 2

C. 3

D 4

Type Invariants

Quiz

```
package Counter_Package is
   type Counter T is private:
   procedure Increment (Val : in out Counter_T);
private
   function Check Threshold (Value : Integer)
                                 return Boolean:
   type Counter_T is new Integer with
      Type Invariant => Check Threshold
                        (Integer (Counter T)):
end Counter_Package;
package body Counter Package is
   function Increment_Helper (Helper_Val : Counter_T)
                                return Counter T is
      Next Value : Counter T := Helper Val + 1:
   begin
      return Next_Value;
   end Increment Helper:
   procedure Increment (Val : in out Counter_T) is
   begin
      Val := Val + 1;
      Val := Increment Helper (Val);
   end Increment;
   function Check_Threshold (Value : Integer)
                                     return Boolean is
      (Value <= 100); -- check against constraint
end Counter_Package;
```

If **Increment** is called from outside of Counter_Package, how many times is **Check_Threshold** called?

- A. 1
- в. **2**
- **C**. 3
- **D**. 4

Type Invariants are only evaluated on entry into/exit from externally visible subprograms. So Check_Threshold is called when entering/exiting Increment - not Increment_Helper

Subtype Predicates

Subtype Predicates Concept

- Ada defines support for various kinds of constraints
 - Range constraints
 - Index constraints
 - Others...
- Language defines rules for these constraints
 - All range constraints are contiguous
 - Matter of efficiency
- Subtype predicates generalize possibilities
 - Define new kinds of constraints

Predicates

- Something asserted to be true about some subject
 - When true, said to "hold"
- 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

Really, type and subtype Predicates

- Applicable to both
- Applied via aspect clauses in both cases
- Syntax

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

(Sub)Type 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 Checks Placement

- Anywhere value assigned that may violate target constraint
- Assignment statements
- Explicit initialization as part of object declaration
- Subtype conversion
- Parameter passing
 - All modes when passed by copy
 - Modes in out and out when passed by reference
- Implicit default initialization for record components
- On default type initialization values, when taken

References Are Not Checked

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Even_Number_Test is
  subtype Even is Integer with Dynamic_Predicate => Even mod 2 = 0;
  Current_Value, Next_Value : Even;
begin
  -- predicates are not checked here
  Put_Line ("Current_Value is" & Current_Value'Image);
  Put_Line ("Next_Value is" & Next_Value'Image);
  -- predicate is checked here
  Current_Value := Next_Value; -- assertion failure here
  Put_Line ("Current_Value is" & Current_Value'Image);
  Put_Line ("Current_Value is" & Current_Value'Image);
  Put_Line ("Next_Value is" & Next_Value'Image);
  end Even_Number_Test;
```

Output would look like

```
Current_Value is 1969492223
Next_Value is 4220029
```

raised SYSTEM.ASSERTIONS.ASSERT_FAILURE: Dynamic_Predicate failed at even_number_test.adb:9

Predicate Expression Content

Reference to value of type itself, i.e., "current instance"

subtype Even is Integer
with Dynamic_Predicate => Even mod 2 = 0;
Current_Value, Next_Value : 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

Allowed Static Predicate Content (1)

- Ordinary Ada static expressions
- Static membership test selected by current instance
- Example 1

Example 2

type Days is (Sun, Mon, Tue, Wed, Thu, Fri, Sat); -- only way to create subtype of non-contiguous values subtype Weekend is Days with Static_Predicate => Weekend in Sat | Sun; AdaCore 897/928

```
Type Contracts
```

Allowed Static Predicate Content (2)

 Case expressions in which dependent expressions are static and selected by current instance

```
type Days is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
subtype Weekend is Days with Static_Predicate =>
  (case Weekend is
   when Sat | Sun => True,
   when Mon .. Fri => False);
```

Note: if-expressions are disallowed, and not needed

```
subtype Drudge is Days with Static_Predicate =>
    -- not legal
    (if Drudge in Mon .. Fri then True else False);
-- should be
subtype Drudge is Days with Static_Predicate =>
    Drudge in Mon .. Fri;
```

Allowed Static Predicate Content (3)

- A call to =, /=, <, <=, >, or >= where one operand is the current instance (and the other is static)
- Calls to operators and, or, xor, not
 - Only for pre-defined type Boolean
 - Only with operands of the above
- Short-circuit controls with operands of above
- Any of above in parentheses

```
Type Contracts
```

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' | '0' | 'U' => True,
   when others => False); -- evaluated at run-time
```

- Plus calls to functions
 - User-defined
 - Language-defined

```
Type Contracts
```

Types Controlling For-Loops

Types with dynamic predicates cannot be used

Too expensive to implement

```
subtype Even is Integer
with Dynamic_Predicate => Even mod 2 = 0;
...
-- not legal - how many iterations?
for A_Number in Even loop
...
end loop;
```

Types with static predicates can be used

```
type Days is (Sun, Mon, Tues, Wed, Thu, Fri, Sat);
subtype Weekend is Days
with Static_Predicate => Weekend in Sat | Sun;
-- Loop uses "Days", and only enters loop when in Weekend
-- So "Sun" is first value for A_Day
for A_Day in Weekend loop
...
```

end loop;

AdaCore

Why Allow Types with Static Predicates?

```
    Efficient code can be generated for usage

  type Days is (Sun, Mon, Tues, We, Thu, Fri, Sat);
 subtype Weekend is Days with Static Predicate => Weekend in Sat | Sun:
 for A Day in Weekend loop
   GNAT.IO.Put Line (A Day'Image);
 end loop:
for loop generates code like
 declare
   a day : weekend := sun;
 begin
   loop
      gnat io put line 2 (a day'Image);
      case a day is
        when sun =>
          a dav := sat:
        when sat =>
         exit:
        when others =>
          a day := weekend'succ (a day);
      end case:
   end loop:
 end;
```

AdaCore

In Some Cases Neither Kind Is Allowed

- No predicates can be used in cases where contiguous layout required
 - Efficient access and representation would be impossible
- Hence no array index or slice specification usage

type Play is array (Weekend) of Integer; -- illegal
type Vector is array (Days range <>) of Integer;
Not_Legal : Vector (Weekend); -- not legal

Special Attributes for Predicated Types

Attributes 'First_Valid and 'Last_Valid

- Can be used for any static subtype
- Especially useful with static predicates
- 'First_Valid returns smallest valid value, taking any range or predicate into account
- 'Last_Valid returns largest valid value, taking any range or predicate into account
- Attributes 'Range, 'First and 'Last are not allowed
 - Reflect non-predicate constraints so not valid
 - Range is just a shorthand for 'First .. 'Last
- **Succ** and **'Pred** are allowed since work on underlying type

Initial Values Can Be Problematic

- Users might not initialize when declaring objects
 - Most predefined types do not define automatic initialization
 - No language guarantee of any specific value (random bits)
 - Example

```
subtype Even is Integer
with Dynamic_Predicate => Even mod 2 = 0;
Some_Number : Even; -- unknown (invalid?) initial value
```

- The predicate is not checked on a declaration when no initial value is given
- So can reference such junk values before assigned
 - This is not illegal (but is a bounded error)

```
Type Contracts
```

Subtype Predicates Aren't Bullet-Proof

 For composite types, predicate checks apply to whole object values, not individual components

```
procedure Demo is
  type Table is array (1 .. 5) of Integer
    -- array should always be sorted
    with Dynamic Predicate =>
      (for all Idx in Table'Range =>
        (Idx = Table 'First or else Table (Idx-1) <= Table (Idx)));
  Values : Table := (1, 3, 5, 7, 9);
begin
  . . .
  Values (3) := 0; -- does not generate an exception!
  . . .
  Values := (1, 3, 0, 7, 9); -- does generate an exception
```

end Demo;

AdaCore

```
Type Contracts
```

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 Index in Sorted_Table'Range =>
   (Index = Sorted_Table'First
        or else Sorted_Table (Index - 1) <= Sorted_Table (Index)));</pre>
```

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;
```

GNAT-Specific Aspect Name Predicate

- Conflates two language-defined names
- Takes on kind with widest applicability possible
 - Static if possible, based on predicate expression content
 - Dynamic if cannot be static
- Remember: static predicates allowed anywhere that dynamic predicates allowed
 - But not inverse
- Slight disadvantage: you don't find out if your predicate is not actually static
 - Until you use it where only static predicates are allowed

Enabling/Disabling Contract Verification

- Corresponds to controlling specific run-time checks
 - Syntax

```
pragma Assertion_Policy (policy_name);
pragma Assertion_Policy (
    assertion_name => policy_name
    {, assertion_name => policy_name});
```

- Vendors may define additional policies (GNAT does)
- Default, without pragma, is implementation-defined
- Vendors almost certainly offer compiler switch
 - GNAT uses same switch as for pragma Assert: -gnata

Quiz

```
type Days T is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
function Is_Weekday (Day : Days_T) return Boolean is
   (Day /= Sun and then Day /= Sat):
Which of the following is a valid subtype predicate?
 A subtype Sub_Day is Days_T with
     Static Predicate => Sub Day in Sun | Sat;
 B. subtype Sub Day is Days T with Static Predicate =>
     (if Sub_Day = Sun or else Sub_Day = Sat then True
   else False);
 C subtype Sub Day is Days T with
     Static Predicate => not Is Weekday (Sub Day);
 D subtype Sub Day is Days T with
     Static Predicate =>
       case Sub Day is when Sat | Sun => True,
              when others => False:
```

Quiz

```
type Days_T is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
function Is_Weekday (Day : Days_T) return Boolean is
  (Day /= Sun and then Day /= Sat);
```

Which of the following is a valid subtype predicate?

```
A subtype Sub Day is Days T with
     Static Predicate => Sub Day in Sun | Sat;
 B subtype Sub Day is Days T with Static Predicate =>
     (if Sub_Day = Sun or else Sub_Day = Sat then True
   else False);
 C subtype Sub Day is Days T with
     Static Predicate => not Is Weekday (Sub Day);
 D subtype Sub Day is Days T with
     Static Predicate =>
       case Sub Day is when Sat | Sun => True,
              when others => False:
Explanations
 A. Correct
 B If statement not allowed in a predicate
 E Function call not allowed in Static Predicate (this would be
   OK for Dynamic_Predicate)
 Missing parentheses around case expression
```

Type Contracts Lab

Overview

- Create simplistic class scheduling system
 - Client will specify name, day of week, start time, end time
 - Supplier will add class to schedule
 - Supplier must also be able to print schedule
- Requirements
 - Monday, Wednesday, and/or Friday classes can only be 1 hour long
 - Tuesday and/or Thursday classes can only be 1.5 hours long
 - Classes without a set day meet for any non-negative length of time

Hints

- Subtype Predicate to create subtypes of day of week
- Type Invariant to ensure that every class meets for correct length of time
- \blacksquare To enable assertions in the runtime from ${\rm GNAT}\ {\rm Studio}$
 - **Edit** \rightarrow Project Properties
 - $\blacksquare \ \textbf{Build} \rightarrow \textbf{Switches} \rightarrow \textbf{Ada}$
 - Click on Enable assertions

AdaCore

```
Type Contracts
```

Type Contracts Lab Solution - Schedule (Spec)

: package Schedule is Maximum Classes : constant := 24; subtype Name T is String (1 .. 10); type Days T is (Mon, Tue, Wed, Thu, Fri, None); type Time T is delta 0.5 range 0.0 .. 23.5; type Classes_T is tagged private; procedure Add Class (Classes : in out Classes T: Name Name T; Days T; Dav Start Time : Time T: End Time : Time T) with Pre => Count (Classes) < Maximum Classes; procedure Print (Classes : Classes T): function Count (Classes : Classes T) return Natural: private 15 subtype Short Class T is Days T with Static Predicate => Short Class T in Mon | Wed | Fri; subtype Long Class T is Days T with Static Predicate => Long Class T in Tue | Thu: type Class_T is tagged record Name : Name T := (others => ' '); Dav : Davs T := None: Start Time : Time T := 0.0: End Time : Time T := 0.0; end record: subtype Class Size T is Natural range 0 .. Maximum Classes: subtype Class Index T is Class Size T range 1 .. Class Size T'Last; type Class Array T is array (Class Index T range ⇔) of Class T; type Classes T is tagged record Size : Class Size T := 0: List : Class Array T (Class Index T); end record with Type Invariant => (for all Index in 1 ... Size => Valid Times (Classes T.List (Index))); function Valid Times (Class : Class T) return Boolean is (if Class.Day in Short Class T then Class.End Time - Class.Start Time = 1.0 elsif Class.Day in Long Class T then Class.End Time - Class.Start Time = 1.5 else Class.End Time >= Class.Start Time); function Count (Classes : Classes T) return Natural is (Classes.Size): 39 end Schedule:

Type Contracts Lab Solution - Schedule (Body)

```
with Ada.Text_IO; use Ada.Text_IO;
   package body Schedule is
3
      procedure Add_Class
4
        (Classes
                   : in out Classes T;
                             Name T:
         Name
         Dav
                             Days_T;
         Start Time :
                             Time T;
         End Time : Time T) is
9
      begin
         Classes.Size
                                     := Classes.Size + 1:
         Classes.List (Classes.Size) :=
12
           (Name
                      => Name, Day => Day,
            Start Time => Start Time, End Time => End Time);
14
      end Add Class:
15
      procedure Print (Classes : Classes T) is
      begin
18
         for Index in 1 .. Classes.Size loop
            Put Line
              (Days T'Image (Classes.List (Index).Day) & ": " &
               Classes.List (Index).Name & " (" &
               Time T'Image (Classes.List (Index).Start Time) & " -" &
               Time_T'Image (Classes.List (Index).End_Time) & " )");
24
         end loop;
      end Print;
26
27
   end Schedule;
28
```

Type Contracts Lab Solution - Main

```
with Ada.Exceptions; use Ada.Exceptions;
   with Ada.Text IO:
                        use Ada.Text_IO;
   with Schedule;
                        use Schedule:
   procedure Main is
      Classes : Classes T:
   begin
      Classes.Add Class (Name
                                    => "Calculus ".
                         Dav
                                    => Mon.
                         Start Time => 10.0.
9
                         End Time
                                    => 11.0):
10
      Classes.Add Class (Name
                                    => "History ".
11
                         Dav
                                    => Tue.
12
                         Start Time => 11.0,
                         End Time => 12.5);
      Classes.Add Class (Name
                                    => "Biology
                                                  ۳.
                         Day
                                    => Wed,
                         Start Time => 13.0,
                         End Time
                                    => 14.0);
18
      Classes.Print:
      begin
                                     => "Chemistry ",
         Classes.Add Class (Name
                                     => Thu,
                            Day
                            Start Time => 13.0,
                            End Time \Rightarrow 14.0);
      exception
         when The Err : others =>
            Put Line (Exception Information (The Err));
      end:
28
   end Main:
29
```

Summary

Working with Type Invariants

- They are not fully 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

Annex - Reference Materials

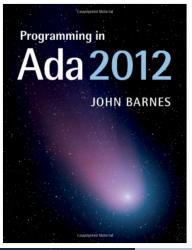
General Ada Information

General Ada Information

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!

General Ada Information

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

GNAT-Specific Help

Annex - Reference Materials

GNAT-Specific Help

Reference Manual

■ Reference Manual(s) available from GNAT STUDIO Help

le Edit Navigate Find Code VCS Build SPARK CodePeer Analyze Debug View Window	Help	
ĝ m 🛛 ○○ + + ◆ ◆ 4 2 2 1 ± ± ▶ £	Welcome	Default search
	Contents	
	GNAT Studio	•
	GNAT Runtime	•
	GNAT	 Native GNAT User's Guide
	GPR	GNAT Reference Manual
	GNU Tools	 Ada 95 Reference Manual
	XMLAda	 Ada 2005 Reference Manual
	Python	 Ada 2012 Reference Manual
	SPARK	Examples
	CodePeer	GNAT User's Guide for Native Platforms
	GNATcoverage	 GNATcheck Reference Manual
	About	GNATstack Reference Manual

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

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.