Ada Essentials

Ada Essentials

Introduction

About AdaCore

The Company

- Founded in 1994
- Centered around helping developers build **safe, secure and reliable** software
- Headquartered in New York and Paris
  - Representatives in countries around the globe
- Roots in Open Source software movement
  - Provide toolchains for Ada/SPARK, C/C++ and Rust
  - Focus on safety-critical and mission-critical systems

About This Training

Your Trainer

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

Goals of the Training Session

- **Build Foundational Confidence:** Feel confident about your basic understanding of the language
- **Learn How to Learn:** Gain the skills to find information and solve new problems
- **Embrace the Process:** Understand that this course is one of many steps in your learning journey
- 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
- Your experience in software development
- 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
  - Class reflection after some labs
- Recommended Setup
  - GNAT Pro Ada
  - GNAT Studio

Styles

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

Warning
This is a warning

Note
This is an important piece of info

Tip
This is a tip

A Note about Syntax

- We use Backus-Naur Form (BNF) to show syntax for many constructs
  - BNF for some constructs can be comprehensive
  - Shows a lot more than we need for this class
- BNF in this course may be simplified to focus only on our needs
  - Elements missing or renamed
  - Full BNF is available in the Reference Manual

Overview

A Little History

The Name

- First called DoD-1
- Augusta Ada Byron, "first programmer"
    - Lord Byron's daughter
    - Planned to calculate **Bernouilli's numbers**
    - **First** computer program
    - On **Babbage's Analytical Engine**
- International Standards Organization standard
    - Updated about every 10 years
- It's **Ada**, not **ADA** (not an acronym)

Ada Evolution Highlights

Ada 83 Abstract Data Types
Modules
Concurrency
Generics
Exceptions

Ada 95 OOP
Child Packages
Annexes

Ada 2005 Multiple Inheritance
Containers
Ravenscar

Ada 2012 Contracts
Iterators
Flexible Expressions

Ada 2022 `'Image` for all types
Declare expression

Note

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

Big Picture

Core Language Content

All compilers/run-times support these "core concepts" of Ada

- Types *(Language- and user-defined)*
- Subprograms (`function` *and* `procedure`)
- Packages *(grouping related entities)*
- Generic Units *(code templates)*
- Language-Based Concurrency *(multi-tasking)*
- Exceptions *(handling unexpected errors)*
- Dynamic memory management
- Object-Oriented Programming
- Contract-Based Programming
- Low Level Programming
- Mixed-language applications

Specialized Needs Annexes

In addition to the "core concepts", Ada compilers can also support

- Real-Time Systems
  - Multi-tasking issues such as priority and timing
- Distributed Systems
  - Multiple partitions as part of a single Ada program
- Numerics
  - Complex arithmetic, improved floating point accuracy, and very large numbers
- High-Integrity Systems
- Information systems

Note

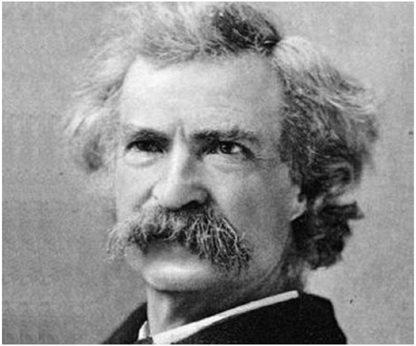
They implement no additional syntax, and may be present or not depending on the compiler/run-time

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*



Hello World!

Introducing Ada

Canonical First Program

"Hello World" - written in Ada!

```
with Ada.Text_IO;
--  Everyone's first program
procedure Hello_World is
begin
   Ada.Text_IO.Put_Line ("Hello, World");
end Hello_World;
```

Canonical First Program

"Hello World" - written in Ada!

```
with Ada.Text_IO;
--  Everyone's first program
procedure Hello_World is
begin
   Ada.Text_IO.Put_Line ("Hello, World");
end Hello_World;
```

- `with` - package dependency (similar to `import` or `#include`)

Canonical First Program

"Hello World" - written in Ada!

```
with Ada.Text_IO;
--  Everyone's first program
procedure Hello_World is
begin
   Ada.Text_IO.Put_Line ("Hello, World");
end Hello_World;
```

- `with` - package dependency (similar to `import` or `#include`)
- `--` - Comment (always goes to end of line)

Canonical First Program

"Hello World" - written in Ada!

```
with Ada.Text_IO;
--  Everyone's first program
procedure Hello_World is
begin
   Ada.Text_IO.Put_Line ("Hello, World");
end Hello_World;
```

- `with` - package dependency (similar to `import` or `#include`)
- `--` - Comment (always goes to end of line)
- `procedure` - subprogram declaration (name of subprogram is `Hello_World`)

Canonical First Program

"Hello World" - written in Ada!

```
with Ada.Text_IO;
--  Everyone's first program
procedure Hello_World is
begin
   Ada.Text_IO.Put_Line ("Hello, World");
end Hello_World;
```

- `with` - package dependency (similar to `import` or `#include`)
- `--` - Comment (always goes to end of line)
- `procedure` - subprogram declaration (name of subprogram is `Hello_World`)
- `begin` - used to start a block of statements

Canonical First Program

"Hello World" - written in Ada!

```
with Ada.Text_IO;
--  Everyone's first program
procedure Hello_World is
begin
   Ada.Text_IO.Put_Line ("Hello, World");
end Hello_World;
```

- `with` - package dependency (similar to `import` or `#include`)
- `--` - Comment (always goes to end of line)
- `procedure` - subprogram declaration (name of subprogram is `Hello_World`)
- `begin` - used to start a block of statements
- `Ada.Text_IO.Put_Line` is a subprogram that prints a string (it's defined in the package we specified on line 1)

Canonical First Program

"Hello World" - written in Ada!

```
with Ada.Text_IO;
--  Everyone's first program
procedure Hello_World is
begin
   Ada.Text_IO.Put_Line ("Hello, World");
end Hello_World;
```

- `with` - package dependency (similar to `import` or `#include`)
- `--` - Comment (always goes to end of line)
- `procedure` - subprogram declaration (name of subprogram is `Hello_World`)
- `begin` - used to start a block of statements
- `Ada.Text_IO.Put_Line` is a subprogram that prints a string (it's defined in the package we specified on line 1)
- `end` - used to end a block of statements. It's optional to add the name of the block you are ending

Lab

Hello World Lab

- This lab focuses more on running labs and verifying your setup
    - Goal for this lab is to write an application that says `Hello, World`
- Almost all our labs are the same. In each lab, we will
    - Copy the appropriate `prompt` directory
    - Open the project that you just copied over
    - Make the changes necessary to perform the lab
    - Compile and run the lab
    - Compare the actual results to the expected results
    - Refer to the `answer` directory if you need help
- This particular lab will walk you through those steps

Step 1 - Copy the Prompt

- You should have downloaded (or received) a `labs` folder
  - This folder contains a directory for each module in the class that has a lab
- In each module directory, you should find a `prompt` and `answer` folder
- The `prompt` folder contains
  - Project file (`default.gpr`) that instructs the compilation system on how to build the application
  - Ada file(s) containing the source code for the application
- Create a folder for this class. As you do each lab, copy the module directory into the class folder

Step 2 - Open the Project

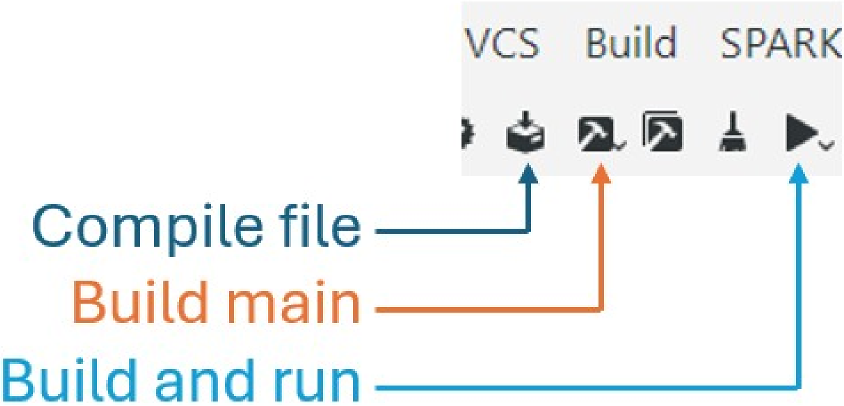
Using GNAT Studio

- If GNAT Studio is on your path, you can either
  - Double-click the `default.gpr` project file
  - Run `gnatstudio default.gpr` file from a command prompt in the correct folder
- Open the GNAT Studio application
  - File → Open Project to open the project file

Step 3 - Make the Necessary Changes

- In the left pane (**Project** tab if it not selected), expand the triangles until you see `hello_world.adb` - double-click it
- In our example, we want to follow the "prompt" comment on line 5
  - On line 6, replace `<something>` with `Hello, World`

Step 4 - Compile and Run the Lab



- After execution, search for the Run tab on the **Messages** window

Step 5 - Compare Actual to Expected

- If the actual results match the goal of the lab
  - Congratulations - you've done it!
  - In this lab, we should see `Hello, World` in the **Run** tab
- If they don't, go back to Step 3
  - Or go on to Step 6

Step 6 - Looking at the Answer

- In the `answer` folder will be the source code for **a** correct solution
  - Look at the part you think is most likely wrong
    - Then go to Step 3 and see if that hint helps
  - Continue until you get the expected result
    - Even if that means copying the whole answer so you can understand it
- Even when you got it right yourself, looking at the answer may give you another method of solving the problem

Summary

First Steps in Writing Ada

- **Hello, World** in Ada is just as simple as any other language
  - Semantics and Syntax may be different, but concepts are similar
- We've taken the first step in speaking about some common concepts in Ada's own language
  - Comments
  - Subprograms
  - Code blocks
  - Input/Output (OK, only output so far!)
- On to the real work!

Type Model

What Is A Type?

Definiton of a Type

- A *type* classifies values and tells the compiler/interpreter
  - What they mean
  - How to use them
- **Type** = label + rules
  - What kind of data (e.g., number, text, boolean)
  - What operations are allowed (e.g., addition, comparison)
- Examples
  - Rust: `u64`, `isize`
  - C: `int`, `char *`
  - Ada: `Integer`, `Boolean`

Note

A **type** is a blueprint for data; you can't mistake the blueprint for a bicycle with the blueprint for a car

Real Job of a Type System

Types serve multiple critical roles

- Validation
    - Catch errors before they happen
- Documentation
    - Make code more readable and self-explanatory
- Optimization
    - Help the compiler produce efficient code
- Abstraction
    - Allow complex operations to be packaged cleanly

Types in Ada

Ada's Strong Typing Model

- Ada is *strongly* and *statically* typed
  - Types are checked at compile-time, not run-time
- Every object has a specific type
  - Explicit conversions of similar types are allowed
- Type safety is a core design goal
  - Prevents accidental operations between incompatible types

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
```

Type System Spectrum

Language	Static Typing	Strong Typing	Implicit Conversion
Ada	✓	✓ (very)	X
C/C++	✓	X	✓
Python	X	✓	✓
Rust	✓	✓	X
Java	✓	✓ (mostly)	X
JavaScript	X	X	✓

Type Model Run-Time Costs

- Checks at compilation **and** run-time
- Good code requires ranges to be verified
  - By user writing the checks **OR**
  - By compiler inserting them
    - Sometimes compiler can even flag failures

C

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

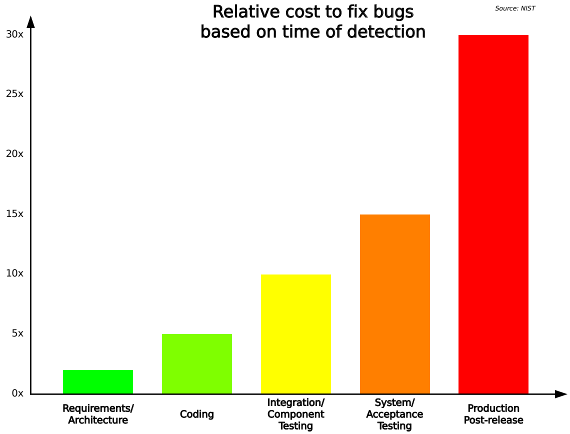
Ada

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

> **Note**
> Checks need to be made, so performance shouldn't be affected by how the checks are inserted

The Type Model Saves Money

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



Ada Types

Ada "Named Typing"

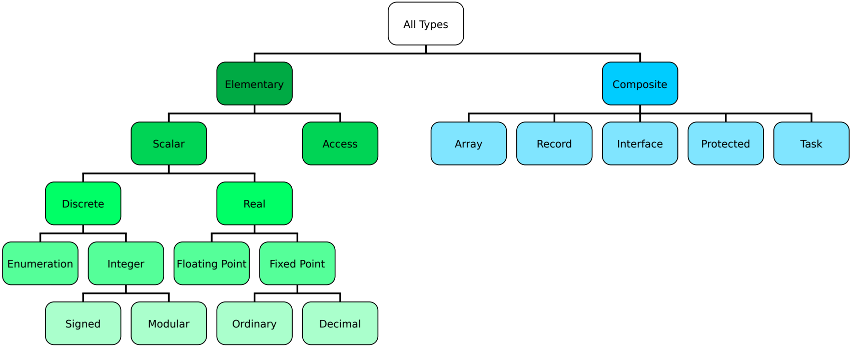
- **Name** differentiates types
- Structure does **not**
- Identical structures may **not** be interoperable

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

Note

In Ada, types are like airport security: if your bag doesn't match the rules, you're not getting through

Categories of Types



Understanding Types vs Subtypes

- **Type** defines a distinct set of values and operations
- *Subtype* (usually) restricts the range of values from a base type
  - Doesn't define a new type

    ```
    type Temperature is range -273 .. 5000;
    subtype Celsius is Temperature range -273 .. 100;
    ```

- Subtype that does **not** add a restriction is generally refered to as an *alias*

```
subtype Water_Temperature is Temperature;
```

Note

Subtypes are Ada's way of saying, "Yes, but not all the values, please."

Declarations

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

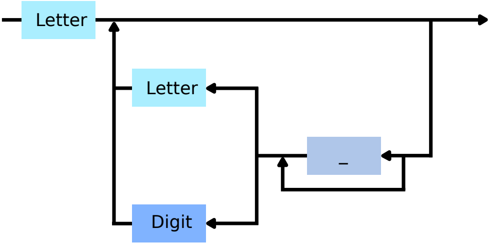
- *Declaration* associates an *identifier* to an *entity*
  - Objects
  - Types
  - Subprograms
  - et cetera
- In a *declarative part*
- Example: `Something : Typemark := Value;`
  - `Something` is an *identifier*
- **Some** implicit declarations
  - **Standard** types and operations
  - **Implementation**-defined

⚠ **Warning**

Declaration **must precede** use

Identifiers and Comments

Identifiers



- Legal identifiers
  ```
  Phase2
  A
  Space_Person
  ```
- Not legal identifiers
  ```
  Phase2__1
  A_
  _space_person
  ```

⚠ **Warning**

Reserved words are **forbidden**

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

Identifiers vs Names

- **Identifier**
  - Syntactic form used typically to introduce entities when declared

```
type RecordT is
  Field : Types.SmallT;
end record;
Str   : String := "Hello;"
Pos   : Integer := Character'val(Str(2));
Index : Types.SmallT := Types.SmallT(Pos);
Rec   : RecordT := (Field => Index);
```

- **Name**
  - Starts with an identifier
  - Can be followed by one or more suffixes
    - Indicates something specific, such as a record component or an array index

```
type RecordT is
  Field : Types.SmallT;
end record;
Str   : String := "Hello;"
Pos   : Integer := Character'val(Str(2));
Index : Types.SmallT := Types.SmallT(Pos);
Rec   : RecordT := (Field => Index);
```

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
```

Literals

Numeric Literals

Syntax

```
decimal_literal ::= numeral [.numeral] [exponent]
numeral ::= digit {[underline] digit}
exponent ::= E [+] numeral | E - numeral
```

Tip

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

- **E** (exponent) must always be an integer
- Examples

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

Based Numeric Literals

Syntax

```
based_literal ::=
   base # based_numeral [.based_numeral] # [exponent]
base ::= numeral
based_numeral ::=
   extended_digit {[underline] extended_digit}
```

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

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

Comparison to C's Based Literals

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

Quiz

Which one of the below is a valid numeric literal?

A. `0_1_2_3_4`
B. `12.`
C. `8#77#E+1.0`
D. `2#1111`

Quiz

Which one of the below is a valid numeric literal?

A. *`0_1_2_3_4`*
B. `12.`
C. `8#77#E+1.0`
D. `2#1111`

Explanations

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

Object Declarations

Object Declarations

Syntax

```
object_declaration ::=
   identifier_list : [constant] typemark [:= expression];
identifier_list ::= identifier {, identifier}
```

- An object is either *variable* or *constant*
  - where
    - <**identifier**> is the defining name for the object
    - <**typemark**> is the name describing the type of the object
- Constant should have a value
  - Except for privacy (seen later)
- Examples

```
An_Object : Some_Type;
Max : constant Some_Type := 200;
-- variable with a constraint
Count : Some_Type range 0 .. Max := 0;
-- dynamic initial value via function call
Some_Object : Some_Type := Some_Function (Count);
```

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 : Some_Type := 10;
  Next_One : Some_Type := First_One;
  Another_One : Some_Type := Next_One;
begin
  ...
```

Multiple Object Declarations

- Allowed for convenience

```
Val_1, Val_2 : Some_Type := Next_Available (Some_Num);
```

- Identical to series of single declarations

```
Val_1 : Some_Type := Next_Available (Some_Num);
Val_2 : Some_Type := Next_Available (Some_Num);
```

> ⚠ **Warning**
>
> May get different value!
>
> ```
> T1, T2 : Time := Current_Time;
> ```

Predefined Declarations

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

Implicit Vs Explicit Declarations

- *Explicit* → in the source

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

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

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

  - Compiler creates appropriate operators based on the underlying type
    - **Numerics** - standard math operators
    - **Arrays** - concatenation operator
    - **Most types** - assignment operator

Named Numbers

Named Numbers

Syntax

```
identifier ::= constant := static_expression;
```

- Associate an **identifier** with a **mathematical expression**
  - Used as **constant**
  - Compatible with integer / real
  - Expression must be **static**
- Examples

```
Pi : constant := 3.141592654;
One_Third : constant := 1.0 / 3.0;
Radians_In_Circle : constant := 2.0 * Pi;
```

Named Number Benefit

- Named numbers are exact — they're not limited by a type's range or precision
- Evaluation at **compile time**

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

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

Scope and Visibility

Introducing Block Statements

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

Syntax

```
block_statement ::=
   [block_statement_identifier:]
      [declare
          declarative_part]
      begin
         sequence_of_statements
      end [block_identifier];
```

Example

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

Scope and "Lifetime"

- Object in scope $\rightarrow$ exists while its enclosing block exists

> **Note**
>
> No *scoping* keywords (C's **static**, **auto** etc...)

```
Outer_Block : declare
   Outer : Integer;
begin
   Outer := 1;
   Inner_Block : declare
      Inner : Float;
   begin
      Inner := 1.0;
   end Inner_Block;
   Outer := Outer + 1;
end Outer_Block;
```

Scope of Outer

Scope of Inner

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

Visibility in Action

- **Name hiding**: a name used in an *inner scope* can hide the same name visible in the *outer scope*

```
type Color is (Red, Green, Blue);
type Size  is (Small, Medium, Large);

declare
  My_Obj : Color;       -- outer My_Obj
begin
  My_Obj := Green;      -- assigns to outer My_Obj (Color)
  declare
    My_Obj : Size;      -- hides outer My_Obj
  begin
    My_Obj := Medium; -- OK: inner My_Obj is Size
    My_Obj := Red;      -- compile error: inner My_Obj is not Color
  end;
  My_Obj := Blue;       -- OK: outer My_Obj is Color
  My_Obj := Small;      -- compile error: outer My_Obj is not Size
end;
```

Overcoming Hiding

- Add a **prefix**
  - Needs named scope

⚠ Warning

- Repeated name reuse is an indication of a *bigger problem*
  - May need refactoring...

```
type Color is (Red, Green, Blue);
type Size  is (Small, Medium, Large);

Outer : declare
  My_Obj : Color;
begin
  My_Obj := Green;         -- outer (Color)
  declare
    My_Obj : Size;         -- inner (Size) hides the outer one
  begin
    My_Obj := Small;       -- inner Size
    Outer.My_Obj := Blue; -- apply prefix to use the hidden Color
  end;
end Outer;
```

Quiz

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

```
declare
   Value : Some_Type := 1;
begin
   Value := Value + 1;
   declare
      Value : Some_Type := 2;
   begin
      Value := Value + 2;
      Print (Value);
   end;
   Print (Value);
end;
```

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

Quiz

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

```
declare
   Value : Some_Type := 1;
begin
   Value := Value + 1;
   declare
      Value : Some_Type := 2;
   begin
      Value := Value + 2;
      Print (Value);
   end;
   Print (Value);
end;
```

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

Explanation

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

Aspects

Pragmas

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

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

  - Inlining of code

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

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

    ```
    pragma My_Own_Pragma;

    - **No effect**
    - Cause **warning** (standard mode)
    ```

  - Must follow correct syntax

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

⚠ Warning

Malformed pragmas are **illegal**

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

Aspect Clauses

Syntax

```
aspect_specification ::=
   with aspect_mark [=> aspect_definition] {,
        aspect_mark [=> aspect_definition] }
```

- 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

Note

Aspect clauses always part of a **declaration**

Aspect Clause Example: Objects

Updated object syntax

```
object_declaration ::=
   identifier_list : [constant] typemark
      [:= expression] [aspect_specification];
```

Example

```
-- 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** values only
- Longhand

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

- Aspect name only → **True**

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

- No aspect → **False**

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

Summary

Summary

- Defines a **single** type, permanently
- Named-numbers
- **Elaboration** concept
  - Value and memory initialization at **run-time**
- Simple **scope** and **visibility** rules
  - **Prefixing** solves **hiding** problems
- Detailed syntax definition in Annex P (using BNF)

Scalar Types

Introduction

Discrete Types

- **Individual** ("discrete") values
  - Can easily identify next/previous value
- Integer types
  - Signed integer types

    ```
    type Integer_T is range -1 .. 1000;
    Number : Integer_T := 123;
    ```

  - Modular integer types
    - No sign bit
    - **Wrap-around** semantics
    - Bitwise operations

    ```
    type Unsigned_T is mod 256;
    Unsigned : Unsigned_T := 123;
    Bitwise  : Unsigned_T := Unsigned and 16#55#;
    ```

- Enumeration types
  - Ordered list of **logical** values

```
type Enumeration_T is (Red, Yellow, Green);
```

Real Types

- *Floating-point* numbers have **variable** exponent portion
  - Allows for a very wide range of values
- *Fixed-point* numbers have a **constant** exponent portion
  - Allows for simpler (integer-based) computer math

```
type Float_T is digits 6;
type Fixed_T is delta 0.01;
```

Discrete Numeric Types

Signed Integer Types

Syntax

```
signed_integer_type_definition ::=
   type identifier is
      range static_simple_expression .. static_simple_expression;
```

- Range of signed **whole** numbers
  - Symmetric about zero (-0 = +0)
- Implicit numeric operators

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

Signed Integer Bounds

- Bounds must be known at compile time

  Later we'll learn about scoping and how to define dynamic bounds

- Compiler selects **base type**
  - Hardware-supported integer type

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

  - Compiler could select a 16-, 32-, or 64-bit representation
    - But not 8-bit, because it won't fit
  - Compilation **error** if not possible

```
3 type Big_Integer_T is range 0 .. 2**128;
```

```
main.adb:3:26:  error:  integer type definition bounds out of range
```

Predefined Signed Integer Types

- `Integer` >= **16 bits** wide
- Other **probably** available
  - `Long_Integer`, `Short_Integer`, etc.
  - Guaranteed ranges: `Short_Integer <= Integer <= Long_Integer`
  - Ranges are all **implementation-defined**

⚠ Warning

Portability not guaranteed

- But 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`

Note

Exponentiation (`**`) result will be a signed integer

- Power **must** be `Integer` `>= 0`

Signed Integer Overflows

- Finite binary representation
- Common source of bugs

```
K : Short_Integer := 16#7FFF#;
...
K := K + 1;

 2#0111_1111_1111_1111#  = (2**16)-1
+                     1
=======================
 2#1000_0000_0000_0000#  = -32,768
```

Signed Integer Overflow: Ada Vs Others

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

Modular Types

Syntax

```
modular_type_definition ::=
   type identifier is mod static_expression;
```

- Integer type
- **Unsigned** values
- Adds operations and attributes

> **Note**
> Typically **bit-wise** manipulation

- 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** on overflow
  - Like other languages' unsigned types
- Additional bit-oriented operations are defined
  - `and`, `or`, `xor`, `not`
  - **Bit shifts**
  - Values as **bit-sequences**

Predefined Sized Numeric 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;
```

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 the value -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 the value -10***
D. Unknown - depends on the compiler

Explanations

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

Attributes

What is an Attribute?

- 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

Image Attribute

- One of the most common attributes is `'Image`
    - Convert an object to a string representation
- Originally treated like a subprogram to convert scalar objects

```
Typemark'Image (Scalar_Object)
```

- Ada 2012 added the ability to use the attribute directly

```
Scalar_Object'Image
```

- Ada 2022 added the ability to use the attribute on non-scalar objects

```
Any_Object'Image
```

Attributes for All Numeric Types

```
type Signed_T is range -99 .. 100;
```

- `T'First`
  - First (**smallest**) value of type `T`
  - `Signed_T'First` → **-99**
- `T'Last`
  - Last (**greatest**) value of type `T`
  - `Signed_T'Last` → **100**
- `T'Range`
  - Shorthand for `T'First .. T'Last`
  - `Signed_T'Range` → **-99 .. 100**
- `T'Min (Left, Right)`
  - **Lesser** of two values of type `T`
  - `Signed_T'Min (12, 34)` → **12**
- `T'Max (Left, Right)`
  - **Greater** of two values of type `T`
  - `Signed_T'Max (12, 34)` → **34**

Enumeration Types

Enumeration Types

Syntax

```
enumeration_type_definition ::=
  type identifier is (enumeration_literal_specification
                        {, enumeration_literal_specification});
enumeration_literal_specification ::=
   defining_identifier | defining_character_literal
```

- Enumeration of **logical** values
  - Integer value is an implementation detail
- Literals
  - Distinct, ordered
  - Can be in **multiple** enumerations

Examples

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

Enumeration Type Operations

- Assignment, relationals
- **Not** numeric quantities

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

Enumeration Type Attributes

```
type Days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
```

- `T'Pred (Input)`
  - Predecessor of specified value
  - `Days'Pred (Tue)` → `Mon`
- `T'Succ (Input)`
  - Successor of specified value
  - `Days'Succ (Tue)` → `Wed`
- Additional information
  - Going past the end (`Days'Pred(Mon)` or `Days'Succ(Sun)`) raises `Constraint_Error`
  - These attributes are available for all scalars, but not particularly useful for numerics

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
```

Boolean Operators' Operand Evaluation

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

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

Short-Circuit Control Forms

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

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

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

Quiz

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

Which statement(s) is (are) legal?

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

Quiz

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

Which statement(s) is (are) legal?

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

Explanations

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

Real Types

Real Types

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

Real Type (Floating and Fixed) Literals

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

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

-- floating point with range
type Percentage is digits 7 range 0.0 .. 100.0;
Valid_Score   : Percentage := 95.5;
Bad_Score     : Percentage := -10.0;  -- runtime error
```

Declaring Floating Point Types

Syntax

```
floating_point_definition ::=
   type identifier is
      digits static_expression [real_range_specification];
real_range_specification ::=
   range static_simple_expression .. static_simple_expression
```

- Compiler choses representation
  - From **available** floating point types
  - May be **more** accurate, but not less
  - If none available → declaration is **rejected**
- `System.Max_Digits` - constant specifying maximum digits of precision available for runtime

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

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

Predefined Floating Point Types

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

> **Tip**
>
> It is best, and easy, to **avoid** predefined types
>
> - 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

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

Floating Point Attributes

```
type My_Float is digits 7;
```

- `My_Float'Digits`
  - Number of digits **requested**
  - `My_Float'Digits` → **7**
- `My_Float'Base`
  - Type selected by compiler
- `My_Float'Base'Digits`
  - Number of **actual** digits
  - `My_Float'Base'Digits` → **15**
- `My_Float'Rounding (X)`
  - Integral value nearest to `X`
  - Rounds away from zero
    - `Float'Rounding (0.5) = 1`
    - `Float'Rounding (-0.5) = -1`

Numeric Types Conversion

- Ada's integer and real are *numeric*
  - Holding a numeric value
- Special rule: can always convert between numeric types
  - Explicitly

Warning

`Float` → `Integer` causes **rounding**

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

Quiz

What is the output of this code?

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

A. 7.6E-01
B. Compile Error
C. 8.0E-01
D. 0.0

Quiz

What is the output of this code?

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

A. 7.6E-01
B. Compile Error
C. 8.0E-01
D. ***0.0***

Explanations

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

Miscellaneous

Checked Type Conversions

- Between "closely related" types
  - Numeric types
  - Inherited types
  - Array types
- Illegal conversions **rejected**
  - Unsafe **Unchecked_Conversion** available
- Called as if it was a function
  - Named using destination type name

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

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

Default Value

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

```
type <typemark> 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

Syntax

```
derived_type_definition ::=
   type identifier is new parent_subtype_indication;
parent_subtype_indication ::=
   parent_subtype_mark [constraint]
```

- **identifier** will copy the behavior of **parent_subtype_mark**
- **constraint** optionally adds limitations to the parent type's behavior
- 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

Example

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

Subtypes

Subtype

Syntax

```
subtype_declaration ::=
   subtype defining_identifier is subtype_indication
        [aspect_specification];
subtype_indication ::=
   [null_exclusion] subtype_mark [constraint]
```

- **subtype_mark** *is an existing* `type` *or* `subtype`
- **constraint** can add restrictions to the parent type
- Still the **same** type
  - So no conversion or casting necessary

Note

If no constraint → type alias

Subtype Example

- Enumeration type with **range** constraint

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

- Equivalent to **anonymous** subtype

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

Kinds of Constraints

- Range constraints on scalar types

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

- Other kinds, discussed later
- Constraints apply only to values
- Representation and set of operations are **kept**

Subtype Constraint Checks

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

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

Performance Impact of Constraints Checking

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

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

- Generates assignment checks similar to

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

- These checks can be disabled with `-gnatp`

Optimizations of Constraint Checks

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

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

Range Constraint Examples

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

Quiz

```ada
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

A. 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

Lab

Scalar Types Lab

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

Using the "Prompts" Directory

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

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

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

Scalar 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
```

Scalar Types Extra Credit

See what happens when your data is invalid / illegal

Number of tests = 0

Color type only has one value

Add number larger than 360 to the circle value

Scalar Types Extra Credit

See what happens when your data is invalid / illegal

Number of tests = 0

```
35 Test_Score_Total := Test_Score_Total /
36                     Test_Score_Total_T (Number_Of_Tests);
```

- Compile warning

```
main.adb:35:43:  warning:  division by zero
```

- Runtime error

```
raised CONSTRAINT_ERROR : main.adb:35 divide by zero
```

Color type only has one value

Add number larger than 360 to the circle value

Scalar Types Extra Credit

See what happens when your data is invalid / illegal

Number of tests = 0

```
35 Test_Score_Total := Test_Score_Total /
36                     Test_Score_Total_T (Number_Of_Tests);
```

- Compile warning

```
main.adb:35:43:  warning:  division by zero
```

- Runtime error

```
raised CONSTRAINT_ERROR : main.adb:35 divide by zero
```

Color type only has one value

```
37 Color := Cmyk_T'Pred (Cmyk_T'Last);
```

- Compile error

```
main.adb:37:30:  error:  Pred of "Cmyk_T'First"
```

```
main.adb:37:30:  error:  static expression fails Constraint_Check
```

Add number larger than 360 to the circle value

Scalar Types Extra Credit

See what happens when your data is invalid / illegal

Number of tests = 0

```
35 Test_Score_Total := Test_Score_Total /
36                     Test_Score_Total_T (Number_Of_Tests);
```

- Compile warning

  `main.adb:35:43: warning: division by zero`

- Runtime error

  `raised CONSTRAINT_ERROR : main.adb:35 divide by zero`

Color type only has one value

```
37 Color := Cmyk_T'Pred (Cmyk_T'Last);
```

- Compile error

  `main.adb:37:30: error: Pred of "Cmyk_T'First"`

  `main.adb:37:30: error: static expression fails Constraint_Check`

Add number larger than 360 to the circle value

```
8 type Degrees_T is mod 360;
```

```
36 Angle := Angle + 459;
```

- Compile error

  `main.adb:36:32: error: value not in range of type "Degrees_T" defined at line 8`

Scalar Types Lab Solution - Declarations

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

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

   type Degrees_T is mod 360;

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

   Number_Of_Tests   : Number_Of_Tests_T;
   Test_Score_Total : Test_Score_Total_T;

   Angle : Degrees_T;

   Color : Cmyk_T;
```

Scalar Types Lab Solution - Implementation

```
begin

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

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

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

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

end Main;
```

Summary

Benefits of Strongly Typed Numerics

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

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

User-Defined Numeric Type Benefits

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

Scalar Types

- **Relational** operators defined (<, =, ...)
  - **Ordered**
- Have common **attributes**
- **Discrete** Types
  - Integer
  - Enumeration
- **Real** Types
  - Floating-point
  - Fixed-point

Summary

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

Statements

Introduction

Statement Kinds

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

Tasking-related are seen in the tasking chapter

Procedure Calls (Overview)

- Procedures must be defined before they are called

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

- Procedure calls are statements
  - Traditional call notation

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

  - "Distinguished Receiver" notation

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

- More details in "Subprograms" section

Block Statements

Block Statements

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

Syntax

```
block_statement ::=
   [block_statement_identifier:]
      [declare
          declarative_part]
       begin
          sequence_of_statements
       end [block_identifier];
```

Block Statements Example

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

Null Statements

Null Statements

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

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

Assignment Statements

Assignment Statements

Syntax

```
assignment_statement ::=
   variable_name := 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;
   K : Km_T := 2;
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
  - Input parameters

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

Aliasing the Assignment Target

Ada 2022

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

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

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

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

- Benefit
  - Symbol can be used multiple times in expression

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

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

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

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

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

Quiz

```ada
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);`
E. `B := One_T(Y) + 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);`
E. `B := One_T(Y) + X;`

Explanations

A. Legal - `A` is an untyped constant so it can be used for any integer-based object
B. Legal - `B`, `C` are correctly typed
C. Illegal - No such "+" operator: must convert operand individually
D. Legal - Correct conversion and types
E. Illegal - Even though the right-hand side matches the type, `B` is a constant and cannot be modified

Conditional Statements

If-then-else Statements

- Control flow using Boolean expressions
- Syntax

```
if_statement ::=
    if condition then
      sequence_of_statements
   {elsif condition then
      sequence_of_statements}
   [else
      sequence_of_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

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

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

Case Statements

- Exclusionary choice among alternatives
- Syntax

```
case_statement ::=
   case selecting_expression is
       case_statement_alternative
      {case_statement_alternative}
   end case;

case_statement_alternative ::=
   when discrete_choice_list =>
      sequence_of_statements

discrete_choice_list ::= discrete_choice {'|' discrete_choice}

discrete_choice ::=
   choice_expression | discrete_subtype_indication | range | others
```

Simple "case" Statements

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

Note: No fall-through between cases

Case Statement Rules

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

"When" Block Alternatives

- Single value: `when Tuesday =>`
  - Block is entered when `case` value is `Tuesday`
- Set of values: `when Saturday | Sunday =>`
  - Block is entered when `case` value is either `Saturday` or `Sunday`
- Range of values: `when Tuesday .. Thursday =>`
  - Block is entered when `case` value is between `Tuesday` and `Thursday` inclusive

"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 =>
    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;
```

Dangers of "Others" Case Alternative

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

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

Quiz

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

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

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

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

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

D.
```
end if;
```

Quiz

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

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

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

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

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

D. end if;

Explanations

- A uses the C-style equality/inequality operators
- D is legal because **else** is not required for an **if** block

Quiz

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

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

```
case A is
```

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

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

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

D. `end case;`

Quiz

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

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

```
case A is
```

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

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

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

D. *`end case;`*

Explanations

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

Loop Statements

Basic Loops and Syntax

Syntax

```
loop_statement ::=
   [loop_statement_identifier:]
      [iteration_scheme] loop
         sequence_of_statements
       end loop [loop_identifier];

iteration_scheme ::= while condition
   | for loop_parameter_specification
   | for iterator_specification

loop_parameter_specification ::=
   defining_identifier in [reverse] discrete_subtype_definition
```

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

Example

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

Loop Exit Statements

Syntax

```
exit_statement ::=
   exit [loop_name] [when condition];
```

- Leaves innermost loop
  - Unless loop name is specified
- `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 condition loop
   sequence_of_statements
end loop;
```

- Behaves the same as `loop` with `exit` at the beginning

```
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

Syntax

```
for identifier in [reverse] discrete_subtype_definition
loop
   sequence_of_statements
end loop;
```

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

Example

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

Variable and Sequence of Values

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

```
-- All values of a type or subtype
for Day in Days_T loop
for Day in Days_T range Mon .. Fri 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
```

Low-Level For-loop Parameter Type

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

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

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

- Type is Integer unless otherwise specified

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

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

Null Ranges

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

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

Reversing Low-Level Iteration Direction

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

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

For-Loop Parameter Visibility

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

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

Referencing Hidden Names

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

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

Iterations Exit Statements

Syntax

```
exit_statement ::=
   exit [loop_name] [when condition];
```

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

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

- With name: Specified loop exited

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

For-Loop with Exit Statement Example

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

Quiz

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

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

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

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

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

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

.

Quiz

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

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

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

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

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

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

Explanations

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

.

GOTO Statements

GOTO Statements

Syntax

```
goto_statement ::= goto label;

label ::= <<label_statement_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

- Goal
  - Create a simple program to build a sheet for tracking your daily work schedule
- Requirements
  - For each day of the week, print a line for each hour in a working day
  - Working hours are
    - 8 hours on a regular work day (Monday-Friday)
    - 4 hours on Saturday
    - No work on Sunday
  - If there are no hours to print, write a message instead
- Hints
  - Use a `for` loop to iterate over days of week and hours
  - Use a `case` statement to determine how many hours in a work day
  - Use an `if` statement to determine if you are printing a time or message

Note

For simplicity, feel free to use a 24-hour clock

Statements Lab Solution

```ada
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 is mod 24;
   Start     : Hours;
   Finish    : Hours;
begin
   Day_Loop :
   for Day in Days_Of_Week_T loop
      case Day is
         when Sunday =>
            Start  := 1;
            Finish := 0;
         when Saturday =>
            Start     := 9;
            Finish    := 13;
         when Monday .. Friday =>
            Start     := 9;
            Finish    := 17;
      end case;
      Put_Line (Day'Image);
      Put_Line ("======");
      if Finish < Start then
         Put_Line ("   No work");
      else
         for Hour in Start .. Finish loop
            Put_Line ("   " & Hour'Image & "00");
         end loop;
      end if;
   end loop Day_Loop;
end Main;
```

Summary

Summary

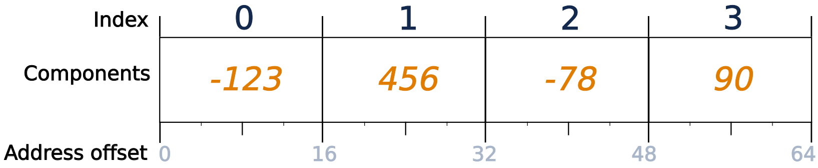
- Assignments must satisfy any constraints of LHS
  - Invalid assignments don't alter target
- Intent to do nothing must be explicitly specified
- Case statements alternatives don't fall through
- Any kind of loop can be expressed with building blocks

Array Types

Introduction

What Is an Array?

- **Definition:** a collection of components
  - ... of the same type
  - ... stored in contiguous memory
  - ... indexed using a discrete range



Array Examples

```
type <typemark> is array (<index_constraint>) of <component_type>;
```

where

- *index_constraint*
    - 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_One is array (1 .. 100) of Integer;

type Discrete_Subtype_Two is range (Able, Baker, Charlie);
type Array_Two is array (Discrete_Subtype_Two) of Float;

type Discrete_Subtype_Three is mod 64;
type Array_Three is array (Discrete_Subtype_Three range 0 .. 31)
   of Interfaces.Integer_16;

type Multidimension_Array is (1 .. 10, 1 .. 10) of Boolean;
```

Arrays in Ada

- Traditional array concept supported to any dimension

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

Array Type Index Constraints

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

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

- Or to constrain unconstrained array types

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

Run-Time Index Checking

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

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

Kinds of Array Types

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

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

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

Constrained Array Types

Constrained Array Type Declarations

Syntax

```
constrained_array_definition ::=
   type identifier is
      array (discrete_subtype_definition
             {, discrete_subtype_definition})
      of subtype_indication;

discrete_subtype_definition ::= discrete_subtype_indication | range
```

> **Note**
> subtype_indication must specify a type whose size is 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;
```

Quiz

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

Which statement(s) is (are) legal?

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

Quiz

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

Which statement(s) is (are) legal?

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

Explanations

A. Legal - components are `Boolean`
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 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

```
unconstrained_array_definition ::=
   type identifier is
      array(index_subtype_definition
            {, index_subtype_definition})
      of subtype_indication;

index_subtype_definition ::= subtype_mark range <>
```

Examples

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

Supplying Index Constraints for Objects

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

- Bounds set by:
  - Object declaration

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

  - Object (or constant) initialization

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

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

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

Bounds Must Satisfy Type Constraints

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

```
type Index is range 1 .. 100;
type Char_Arr is array (Index range <>) of Character;
Good : Char_Arr (50 .. 75);
Bad  : Char_Arr (0 .. 10); -- run-time error
```

```
example.adb:5:21: warning: static value out of range of type "Index" defined at line 2
```

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

```
2  type Index_T is range 1 .. 100; -- Index_T'Size = 8
3
4  type Array_T is array (Index_T range <>) of Integer;
5
6  Typical_Empty_Array : Array_T (1 .. 0);
7  Weird_Empty_Array   : Array_T (123 .. -5);
8  Bad_Empty_Array     : Array_T (999 .. 0);
```

```
example.adb:8:35: error: value not in range of type "Index_T" defined at line 2
```

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

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

```
2 type Component_T is array (Integer range <>) of Boolean;
3 type Good is array (1 .. 10) of Component_T (1 .. 20); -- OK
4 type Bad is array (1 .. 10) of Component_T; -- compile error
```

```
example.adb:4:35: error: unconstrained element type in array declaration
```

Arrays of Arrays

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

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

Quiz

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

Which declaration(s) is (are) legal?

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

Quiz

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

Which declaration(s) is (are) legal?

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

Explanations

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

Strings

"String" Types

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

    ```
    subtype Positive is Integer range 1 .. Integer'Last;
    type String is array (Positive range <>) of Character;
    ```
  - **Wide_String**, with **Wide_Character** as component
  - **Wide_Wide_String**, with **Wide_Wide_Character** as component
    - Ada 2005 and later
- Can be defined by applications too

Application-Defined String Types

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

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

Specifying Constraints Via Initial Value

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

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

Prompt1 : String := "Hello World!";
-- Prompt1'First is Positive'First (1)

type Another_String is array (Integer range <>)
    of Character;

Prompt2 : Another_String := "Hello World!";
-- Prompt2'First is Integer'First
```

String Literals

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

```
-- two double quotes with nothing inside
A_Null_String : constant String := "";

String_Of_Length_One : constant String := "A";

Embedded_Single_Quotes : constant String
                       := "Embedded 'single' quotes";

Embedded_Double_Quotes : constant String
                       := "Embedded ""double"" quotes";
```

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 `O'First` .. `O'Last`

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

Attributes' Benefits

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

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

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

Array 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 (indexes 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)
  - Equality for all scalars
  - Inequality for all discrete types
- Logical (for `Boolean` component type)
- Slicing
  - Portion of array

Slicing

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

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

Result: `We love Ada!`

Example: Slicing with Explicit Indexes

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

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

Idiom: Named Subtypes for Indexes

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

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

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

Dynamic Subtype Constraint Example

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

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

Quiz

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

Which statement(s) is (are) legal?

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

Quiz

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

Which statement(s) is (are) legal?

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

Explanations

A. All objects are just Boolean values
B. 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

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

```
for identifier of [reverse] array_or_container loop
   sequence_of_statements
end loop;
```

- 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;
```

Quiz

```
type Array_T is array (1..5) of Integer
   with Default_Component_Value => 1;
A : Array_T;
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;
```

Which output is correct?

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

Quiz

```
type Array_T is array (1..5) of Integer
   with Default_Component_Value => 1;
A : Array_T;
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;
```

Which output is correct?

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

Explanation

- `Default_Component_Value` so all components intialized to 1
- First `for` loop iterates over indexes `Ada'First + 1` (2) through `Ada'Last - 1` (4) - so array now is 1, 10, 15, 20, 1
- Second `for` loop iterates over whole array backwards (`reverse`) giving the answer of **1 20 15 10 1**

Array Aggregates

Aggregates

- Literals for composite types
    - Array types
    - Record types
- Two distinct forms
    - Positional
    - Named

Aggregate Syntax

```
array_aggregate ::=
    positional_array_aggregate | named_array_aggregate

positional_array_aggregate ::=
    (expression, expression {, expression})
  | (expression {, expression}, others => expression)

named_array_aggregate ::=
    (array_component_association_list)

array_component_association_list ::=
    array_component_association
    {, array_component_association}

array_component_association ::=
    discrete_choice_list => expression
```

Aggregate "Positional" Form

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

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

Aggregate "Named" Form

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

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

Combined Aggregate Forms Not Allowed

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

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

Aggregates Are True Literal Values

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

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

Aggregate Consistency Rules

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

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

"Others"

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

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

Nested Aggregates

- For arrays of composite component types

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

Defaults Within Array Aggregates

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

Syntax

```
positional_array_aggregate ::=
    (expression, expression {, expression})
  | (expression {, expression}, others => expression)
  | (expression {, expression}, others => <>)

array_component_association ::=
    discrete_choice_list => expression
  | discrete_choice_list => <>
```

Examples

```
type Int_Arr is array (1 .. N) of Integer;
Named_Notation      : Int_Arr := (1 => 2, 2 .. N => <>);
Positional_Notation : Int_Arr := (2, 3, 5, others => <>);
```

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);
```

Note

Aggregates for single element arrays must use named notation.

```
type Array_T is array (1..1) of Integer;
Good : Array_T := (1 => 123);
Bad  : Array_T := (456);
```

Quiz

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

Which statement is correct?

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

Quiz

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

Which statement is correct?

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

Explanations

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

Aggregates in Ada 2022

Ada 2022

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

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

  - So common aggregates can use either square brackets or parentheses

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

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

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

  - Single component array

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

Iterated Component Association

Ada 2022

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

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

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

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

  - `Object2` will have each component doubled

More Information on Iterators

Ada 2022

- You can nest iterators for arrays of arrays

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

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

```
Ada2022 : Array_T(1..5) :=
   [for I in 1 .. 2 => -1,
    for J in 4 ..5 => 1,
    others => 0];
```

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

Delta Aggregates

Ada 2022

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

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

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

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

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

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

This works for records as well (see that chapter)

Detour - 'Image for Complex Types

'Image Attribute

Ada 2022

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

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

Yields an output of

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

Overriding the 'Image Attribute

Ada 2022

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

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

Defining the 'Image Attribute

Ada 2022

- Then we need to declare the procedure

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

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

- And then we define it

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

Using the 'Image Attribute

Ada 2022

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

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

  - Generating the following output

```
RED=>TRUE
```

```
YELLOW=>TRUE
```

```
GREEN=>FALSE
```

Anonymous Array Types

Anonymous Array Types

- Array objects need not be of a named type

  ```
  A : array (1 .. 3) of B;
  ```

- Without a type name, no object-level operations
  - Cannot be checked for type compatibility
  - Operations on components are still ok if compatible

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

Lab

Array Lab

- Requirements
  - Create an array type whose index is days of the week and each 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 constant object to part of the non-constant object ("destination")
    - Clear the rest of the non-constant object
    - Print the contents of the non-constant object
- Hints
  - When you want to combine multiple strings (which are arrays!) use the concatenation operator (**&**)
  - Slices are how you access part of an array
  - Use aggregates (either named or positional) to initialize data

Arrays of Arrays

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

Array Lab Solution - Declarations

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

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

   Const_Arr : constant Unconstrained_Array_T :=
                (1, 2, 3, 4, 5, 6, 7);
   Array_Var : Unconstrained_Array_T (Days_Of_Week_T);

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

Array Lab Solution - Implementation

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

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

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

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

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

Summary

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
- Strings are special-case arrays
  - Any single-dimensioned array of some character type is a *string type*
  - Language defines types `String`, `Wide_String`, `Wide_Wide_String`
  - Language-defined support defined in Appendix A - `Ada.Strings`
- Default initialization for large arrays may be expensive!
- Anonymously-typed array objects used in examples for brevity but that doesn't mean you should in real programs

Record Types

Introduction

Syntax and Examples

Syntax

```
record_definition ::=
   type identifier is record
      component_declaration
   end record;

component_declaration ::=
   defining_identifier_list : subtype_indication
      [:= default_expression];
```

Example

```
type Record1_T is record
   Is_Valid : Boolean := False;
   Content  : Integer;
end record;
```

Records can be **discriminated** as well

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

Components Rules

Characteristics of Components

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

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

- **No** constant components

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

- **No** recursive definitions

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

- **No** indefinite types

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

Multiple Declarations

- Multiple declarations are allowed (like objects)

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

- Equivalent to

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

"Dot" Notation for Components Reference

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

- Can reference nested components

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

Quiz

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

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

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

Quiz

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

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

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

A. Anonymous types not allowed
B. Comma-separated list of components is allowed
C. No recursive definition
D. No constant component

Quiz

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

Is the definition legal?

A. Yes
B. No

Quiz

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

Is the definition legal?

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

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

Record Operations

Available Operations

- Predefined
  - Equality (and thus inequality)

    ```
    if A = B then
    ```

  - Assignment

    ```
    A := B;
    ```

- User-defined
  - Subprograms

Assignment Examples

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

Limited Types - Quick Intro

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

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

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

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

Record Aggregates

Aggregates

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

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

Record Aggregate Examples

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

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

Aggregate Completeness

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

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

- Compiler will not catch the missing component

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

- Aggregate must be complete - compiler error

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

Named Associations

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

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

Nested Aggregates

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

Aggregates with Only One Component

Must use named form

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

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

Aggregates with **others**

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

```
2  type Integer_T is new Integer;
3  type Record_T is record
4     A       : Integer_T;
5     B, C, D : Integer;
6  end record;
7
8  Good1 : Record_T := (1, 2, others => 3);
9  Good2 : Record_T := (A => 9, others => 87);
10 Bad   : Record_T := (others => 0);
```

```
example.adb:10:25: error: components in "others" choice must have same type
```

Quiz

```
type Record1_T is record
   Single : Integer;
end record;
type Record2_T is record
   One, Two : Integer;
   Three    : Short_Integer;
   Four     : Record1_T;
end record;

Obj1 : Record1_T;
Obj2 : Record2_T;
```

Which assignment(s) is (are) legal?

A. `Obj2 := (Four => Obj1)`
B. `Obj2 := (Four => Obj1, others => 123)`
C. `Obj2 := (One => 1, Four => Obj1, Three => 3, Two => 2)`
D. `Obj2 := (One => 1, Four => (4), Three => 3, Two => 2)`

Quiz

```
type Record1_T is record
   Single : Integer;
end record;
type Record2_T is record
   One, Two : Integer;
   Three    : Short_Integer;
   Four     : Record1_T;
end record;

Obj1 : Record1_T;
Obj2 : Record2_T;
```

Which assignment(s) is (are) legal?

A. `Obj2 := (Four => Obj1)`
B. `Obj2 := (Four => Obj1, others => 123)`
C. *`Obj2 := (One => 1, Four => Obj1, Three => 3, Two => 2)`*
D. `Obj2 := (One => 1, Four => (4), Three => 3, Two => 2)`

A. Aggregate must be complete - missing values for `One, Two, Three`
B. All fields specified via **`others`** must be of the same type (even if the value is a literal that is allowed for the fields)
C. Legal (order is irrelevant when using named notation)
D. Field `Four` has a single component, so its aggregate must use named notation e.g. `(One => 1, Four => (Single => 4), Three => 3, Two => 2)`

Delta Aggregates

Ada 2022

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

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

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

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

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

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

Note for record delta aggregates you must use named notation

Default Values

Component Default Values

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

Default Component Value Evaluation

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

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

Defaults Within Record Aggregates

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

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

Default Initialization Via Aspect Clause

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

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

Quiz

```
function Next return Natural; -- returns next number (starts at 1)

type Record_T is record
   Height, Width : Integer := Next;
   Color         : Integer := Next;
end record;
Shape : Record_T := (Color => 100, others => <>);
```

What is the value of Shape?

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

Quiz

```
function Next return Natural; -- returns next number (starts at 1)

type Record_T is record
   Height, Width : Integer := Next;
   Color         : Integer := Next;
end record;
Shape : Record_T := (Color => 100, others => <>);
```

What is the value of Shape?

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

Explanations

A. `Color => 100`
B. Multiple declaration calls `Next` twice
C. `Height` is first call to `Next` (1), `Width` is second call to `Next` (2), `Color` initialized to 100
D. `Color => 100` has no effect on `Height` and `Width`

Variant Records

Variant Record Types

- A *discriminated record* uses a special field (*discriminant*) to specify information about the record

```
type Discriminated_Record (Discriminant : Natural) is record
   Text : String (1..Discriminant);
end record;
```

  - All objects of `Discriminated_Record` are of the same type, regardless of the value of `Discriminant`
- A *variant record* is a special case of discriminated record
  - Discriminant is a discrete type
  - Used in a `case` block to control visibility of components
- Kind of *storage overlay*
  - Similar to `union` in C
  - But preserves **type checking**
  - And object size **is related to** discriminant
- Aggregate assignment is allowed

Immutable Variant Record

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

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

> **Note**
> `case` block must be **last** part of the definition - therefore only **one** per 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)
- Discriminant is treated as any other component
  - But is a constant in an immutable variant record

Immutable Variant Record Example

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

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

- Pat has Group, Age, and Gpa
  - Sam has Group, Age, and Pubs
  - Aggregate specifies all components, including the discriminant
- Compiler can detect some problems, but more often clashes are run-time errors

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

  - Pat.Pubs := 3; would generate a compiler warning because compiler knows Pat is a Student
    - warning: Constraint_Error will be raised at run time
  - Do_Something (Pat); generates a run-time error, because only at runtime is the discriminant for Param known
    - raised CONSTRAINT_ERROR : discriminant check failed
- Pat := Sam; would be a compiler warning because the constraints do not match

Mutable Variant Record

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

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

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

Mutable Variant Record Example

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

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

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

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

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

Quiz

```
type Variant_T (Valid : Integer) is record
    case Valid is
    when Integer'First .. -1 =>
        Value : Integer;
        State : Boolean;
    when others =>
        Number : Natural;
    end case;
end record;

Variant_Object : Variant_T (1);
```

Which component(s) does `Variant_Object` contain?

A. `Variant_Object.Value`, `Variant_Object.State`
B. `Variant_Object.Number`
C. None: Compilation error
D. None: Run-time error

Quiz

```
2    type Variant_T (Valid : Integer) is record
3        case Valid is
4        when Integer'First .. -1 =>
5            Value : Integer;
6            State : Boolean;
7        when others =>
8            Number : Natural;
9        end case;
10   end record;
11
12   Variant_Object : Variant_T (1);
```

Which component(s) does `Variant_Object` contain?

A. `Variant_Object.Value`, `Variant_Object.State`
B. ***`Variant_Object.Number`***
C. None: Compilation error
D. None: Run-time error

Explanation

- Variant block covers all possible values of `Valid`, so no compilation error
- Discriminant has a value (1) which is in range, so no run-time error
- `Valid` is 1, so it enters the **`when others`** block on line 7. The block only contains component `Number`.

Quiz

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

Variant_Object : Variant_T (True);
```

Which component does `Variant_Object` contain?

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

Quiz

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

```
Variant_Object : Variant_T (True);
```

Which component does `Variant_Object` contain?

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

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

Lab

Record Types Lab

- Requirements
  - Create a record to measure distance in feet and inches
  - Create two distances in feet and inches
    - Make sure that the inch values when added together will be at least one foot
  - Add the two distances
  - Print all three values
- Hints
  - Feet and inches should be different types
    - Does not makes sense to directly add inches to feet
  - Consider what happens when adding inches overflows

Record Types Lab Solution - Declarations

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

   Max_Feet   : constant := 100;
   Max_Inches : constant := 12;

   type Feet_T is range 0 .. Max_Feet;
   type Inches_T is range 0 .. Max_Inches - 1;

   type Distance_T is record
      Feet   : Feet_T;
      Inches : Inches_T;
   end record;

   Point_1  : Distance_T;
   Point_2  : Distance_T;
   Distance : Distance_T;

   Total : Integer;
```

Record Types Lab Solution - Implementation

```
begin
   Point_1.Feet   := 12;
   Point_1.Inches := 7;

   Point_2 := (Feet   => 6,
               Inches => 8);

   Distance := (0, 0);

   Total := Integer (Point_1.Inches) +
            Integer (Point_2.Inches);
   if Total > Max_Inches then
      Distance.Inches := Inches_T (Total - Max_Inches);
      Distance.Feet   := 1;
   else
      Distance.Inches := Point_1.Inches + Point_2.Inches;
   end if;
   Distance.Feet := Distance.Feet + Point_1.Feet + Point_2.Feet;

   Put_Line ("Point 1: " &
             Point_1.Feet'Image &
             Point_1.Inches'Image);
   Put_Line ("Point 2: " &
             Point_2.Feet'Image &
             Point_2.Inches'Image);
   Put_Line ("Distance: " &
             Distance.Feet'Image &
             Distance.Inches'Image);
end Main;
```

Summary

Summary

- Heterogeneous types allowed for components
- Default initial values allowed for components
  - Evaluated when each object elaborated, not the type
  - Not evaluated if explicit initial value specified
- Aggregates express literals for composite types
  - Can mix named and positional forms
- Variant records allow flexible records that maintain strong typing
  - Immutable records always use the same discriminant value
  - Mutable records can change their discriminant
    - But only when entire object is being assigned

Subprograms

Introduction

Introduction

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

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

- Provide direct syntactic support for separation of specification from implementation

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

Recognizing Procedures and Functions

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

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

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

A Little "Preaching" About Names

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

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

Syntax

Specification and Body

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

Procedure Specification Syntax

Syntax

```
procedure_specification ::=
   ::= procedure subprogram_name parameter_profile

parameter_profile ::=
   [(parameter_specification {; parameter_specification})]

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

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

Examples

```
procedure Swap (A, B : in out Integer);
procedure Clean (Force : Boolean := True);
procedure Reset;
```

Function Specification Syntax

Syntax

```
function_specification ::=
   function function_name parameter_and_result_profile

function_name ::=
   subprogram_name | operator_symbol

parameter_and_result_profile ::=
    [parameter_profile] return subtype_mark
```

Examples

```
function Square (X : Float) return Float;
function Is_Open return Boolean;
```

Body Syntax

Syntax

```
subprogram_specification is
   [declarations]
begin
   sequence_of_statements
end [subprogram_name | operator_symbol];
```

Examples

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

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

Completions

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

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

Completion Examples

- Specifications

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

- Completions

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

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

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

Direct Recursion - No Declaration Needed

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

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

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

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

Indirect Recursion Example

- Elaboration in **linear order**

```
procedure P;

procedure F is
begin
  P;
end F;

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

Quiz

Which profile is semantically different from the others?

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

Quiz

Which profile is semantically different from the others?

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

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

Parameters

Subprogram Parameter Terminology

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

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

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

Parameter Associations in Calls

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

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

- Having named **then** positional is forbidden

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

Parameter Modes

- 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

- Function `return` **must** always be handled
- Return type is **not** an object
  - Type does not have to be constrained

```
function From_String (Value : String) return Integer;
function To_String (Value : Integer) return String;
```

Why Read Mode **out** Parameters?

- No need for readable temporary variable
- Warning: initial value is **not defined**

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

Parameter Passing Mechanisms

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

By-Copy Vs By-Reference Types

- By-Copy
  - Scalar types
  - `access` types
- By-Reference
  - `tagged` types
  - `task` types and `protected` types
  - `limited` types
- `array`, `record`
  - By-Reference when they have by-reference **components**
  - By-Reference for **implementation-defined** optimizations
  - By-Copy otherwise
- `private` depends on its full definition
- Note that the parameter mode `aliased` will force pass-by-reference
  - This mode is discussed in the **Access Types** module

Unconstrained Formal Parameters

- Unconstrained **formals** are allowed
  - Constrained by **actual**

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

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

Unconstrained Return Type

- When a function returns an unconstrained type, the caller needs to be able to handle it

```
function Pad (Length : Natural) return String is
   Padding : String(1..Length) := (others => ' ');
begin
   return Padding;
end Pad;
```

- The client can call Pad to initialize an object, or to assign to an object of the expected size, or pass to another unconstrained parameter

```
declare
   This_Is_OK       : String := Pad (3);
   This_Is_Bad      : String(1..10) := Pad(5);  -- runtime error
   OK_For_Length_4 : String(1..4);
begin
   Put_Line (Pad(50) & "This will always be OK");
   OK_For_Length_4:= Pad (4);  --  Yes, this is OK
   OK_For_Length_4:= Pad (5);  --  No, runtime error
```

Unconstrained Parameters Surprise

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

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

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

Naive Implementation

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

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

Correct Implementation

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

```
function Subtract (Left, Right : Vector) return Vector is
   pragma Assert (Left'Length = Right'Length);

   Result : Vector (Left'Range);
   Offset : constant Integer := Right'First - Result'First;
begin
   for K in Result'Range loop
     Result (K) := Left (K) - Right (K + Offset);
   end loop;

   return Result;
end Subtract;
```

Quiz

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

Which call(s) is (are) legal?

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

Quiz

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

Which call(s) is (are) legal?

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

Explanations

A. P4 is `out`, it **must** be a variable
B. P2 has no default value, it **must** be specified
C. P1 can be a literal, P2 must be an object, P3 can be a literal, P4 must be an object
D. F is a function, its `return` **must** be handled

Null Procedures

Null Procedure Declarations

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

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

- Shorthand form

```
procedure NOP is null;
```

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

Null Procedures As Completions

- Completions for a distinct, prior declaration

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

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

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

Typical Use for Null Procedures: OOP

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

Null Procedure Summary

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

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

Nested Subprograms

Subprograms Within Subprograms

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

Nested Subprogram Example

```
procedure Populate_Lines
   (Lines : in out Types.Lines_T;
    Name  :        String) is

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

begin
   for J in Lines'Range loop
      Lines (J) := Read (J'Image);
   end loop;
end Populate_Lines;
```

Procedure Specifics

Return Statements in Procedures

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

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

Main Subprograms

- Must be library subprograms
  - Not nested inside another subprogram
- No special subprogram unit name required
- Can be many per project
- Can always be procedures
- Can be functions if implementation allows it
  - Execution environment must know how to handle result

```
with Ada.Text_IO;
procedure Hello is
begin
  Ada.Text_IO.Put ("Hello World");
end Hello;
```

Function Specifics

Return Statements in Functions

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

```
function Add (Left, Right : Integer ) return Integer is
begin
   return Left + Right;
end Add;
```

No Path Analysis Required by Compiler

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

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

Multiple Return Statements

- Allowed
- Sometimes the most clear

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

Multiple Return Statements Versus One

- Many can detract from readability
- Can usually be avoided

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

Function Dynamic-Size Results

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

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

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

Expression Functions

Expression Functions

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

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

NB: Parentheses around expression are **required**

- Can complete a prior declaration

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

Expression Functions Example

- Expression function

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

- Is equivalent to

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

Quiz

Which statement is True?

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

Quiz

Which statement is True?

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

Explanation

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

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

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

Potential Pitfalls

Mode **out** Risk for Scalars

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

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

"Side Effects"

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

```
Global : Integer := 0;

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

Order-Dependent Code and Side Effects

```
Global : Integer := 0;

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

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

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

Parameter Aliasing

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

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

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

Functions' Parameter Modes

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

Easy Cases Detected and Not Legal

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

Extended Example

Implementing a Simple "Set"

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

Values for the Set

- Colors of the rainbow

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

- Group of colors

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

- Mechanism indicating which color is in the set

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

Primitive Operations for the Set

- Create a set

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

- Add a color to the set

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

- Remove a color from the set

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

- Check if color is in set

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

Implementation of the Primitive Operations

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

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

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

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

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

Using our Set Construct

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

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

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

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

Lab

Subprograms Lab

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

Subprograms Lab Solution - Search

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

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

Subprograms Lab Solution - Main

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

begin

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

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

end Main;
```

Summary

Summary

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

Type Derivation

Introduction

Type Derivation

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

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 Type Derivation

- Most types can be derived

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

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

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

* :ada:`I_Obj := N_Obj;` |rightarrow| generates a compile error

:color-red:`expected type "Integer_T" defined at line 2`

* But a child can be converted to the parent

* :ada:`I_Obj := Integer_T (N_Obj);`
```

Simple Derivation and Type Structure

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

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

- Unconstrained types can be constrained

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

Primitives

Primitive Operations

- Primitive Operations are those subprograms associated with a type

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

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

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

General Rule for Defining a Primitive

- Primitives are subprograms
- Subprogram `S` is a primitive of type `T` if and only if:
  - `S` is declared in the scope of `T`
  - `S` uses type `T`
    - As a parameter
    - As its return type (for a `function`)
  - `S` is above *freeze-point* (see next section)
- Standard practice
  - Primitives should be declared **right after** the type itself
  - In a scope, declare at most a **single** type with primitives

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

Primitive of Multiple Types

A subprogram can be a primitive of several types

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

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

end P;
```

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

Creating Primitives for Children

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

Overriding Indications

- **Optional** indications
- Checked by compiler

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

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

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

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

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

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

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

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

Quiz

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

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

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

Quiz

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

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

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

B. Infinite recursion (will result in Storage_Error at run-time)
C. Unlike some languages, there is no assignment operator

Freeze Point

What is the "Freeze Point"?

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

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

type Child is new Parent;

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

V : Child;

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

Debugging Type Freeze

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

pkg.ads

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

<obj>/pkg.ads.dg

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

Quiz

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

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

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

type Grandchild is new Child range 3 .. 98;

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

Which are Parent's primitives?

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

Quiz

```ada
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. `Proc_A` appears immediately after type declaration
B. Freeze: `Parent` has been derived
C. Freeze: scope change
D. `Proc_A` is a primitive

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

Introduction

Beyond Simple Expressions

- Different categories of expressions above simple assignment and conditional statements
- Can constrain types to sub-ranges
    - Allows for simple membership checks of values
    - Increases readability and flexibility
- Embedded conditional assignments
    - Equivalent to C's `A ? B : C`
    - ...and even more elaborate!

Membership Tests

"Membership" Operation

Syntax

```
simple_expression [not] in membership_choice_list

membership_choice_list ::= membership_choice
                           { | membership_choice}

membership_choice ::= expression | range | subtype_mark
```

- Acts like a boolean function
- Usable anywhere a boolean value is allowed

Examples

```
X : Integer := ...
B : Boolean := X in 0..5;
C : Boolean := X not in 0..5; -- also "not (X in 0..5)"
```

Testing Constraints Via Membership

```
type Calendar_Days  is
    (Mon, Tues, Wed, Thur, Fri, Sat, Sun);
subtype Weekdays is Calendar_Days range Mon .. Fri;
Day : Calendar_Days := Today;
...
if Day in Mon .. Fri then ...
if Day in Weekdays then ... -- same as above
```

Testing Non-Contiguous Membership

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

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

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

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

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

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

  ```
  * **|** is used to separate members
  * So :ada:`1 | 3 | 5` is the set for which we are verifyi
  ```

Quiz

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

Which condition(s) is (are) legal?

A. `if Today = Mon or Wed or Fri then`
B. `if Today in Days_T then`
C. `if Today not in Weekdays_T then`
D. `if Today in Tue | Thu then`

Quiz

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

Which condition(s) is (are) legal?

A. `if Today = Mon or Wed or Fri then`
B. *`if Today in Days_T then`*
C. *`if Today not in Weekdays_T then`*
D. *`if Today in Tue | Thu then`*

Explanations

A. `Wed` and `Fri` are not Boolean expressions - need to compare each of them to `Today`
B. Legal - should always return `True`
C. Legal - returns `True` if `Today` is `Sat` or `Sun`
D. Legal - returns `True` if `Today` is `Tue` or `Thu`

Qualified Names

Qualification

Syntax

```
qualified_expression ::= subtype_mark'(expression) |
                         subtype_mark'aggregate
```

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

Testing Constraints Via Qualification

- Asserts value is compatible with subtype
  - Raises exception `Constraint_Error` if not true

```
subtype Weekdays is Days range Mon .. Fri;
This_Day : Days;
...
case Weekdays'(This_Day) is -- run-time error if out of range
  when Mon =>
    Arrive_Late;
    Leave_Early;
  when Tue .. Thur =>
    Arrive_Early;
    Leave_Late;
  when Fri =>
    Arrive_Early;
    Leave_Early;
end case; -- no 'others' because all subtype values covered
```

Conditional Expressions

Conditional Expressions

- Ultimate value depends on a controlling condition
- Allowed wherever an expression is allowed
  - Assignment RHS, formal parameters, aggregates, etc.
- Similar intent as in other languages
  - Java, C/C++ ternary operation **A ? B : C**
  - Python conditional expressions
  - etc.
- Two forms:
  - *If expressions*
  - *Case expressions*

If Expressions

Syntax

```
if_expression ::=
   (if condition then dependent_expression
   {elsif condition then dependent_expression}
   [else dependent_expression])
condition ::= boolean_expression
```

- Syntax looks like an *if statement* without **end if**
- The conditions are always Boolean values

```
(if Today > Wednesday then 1 else 0)
```

Result Must Be Compatible with Context

- Conditional expression will be assigned to something
  - Each **branch** of the conditional expression (*dependent expression*) must be of the **same type**
  - Compile error if this is not true

```
Hours_Worked : Float :=
   (if Day_Of_Week in Weekday then 8.0 else 0.0);
--  Hours_Worked will be either 8.0 or 0.0

Modifier : constant String :=
   (if Time < 1200 then "AM"
    elsif Time > 1200 then "PM"
    else "Noon");
--  Modifier will be either AM, PM, or Noon
--  (String lengths are different, but this is initialization)

Bad_Expression : Float :=
   (if Overtime then 1.5 else 1);
```

```
example.adb:21:33: error: type of "else" incompatible with that of "then" expression
```

"If Expression" Example

```
declare
  Remaining : Natural := 5;  -- arbitrary
begin
  while Remaining > 0 loop
    Put_Line ("Warning! Self-destruct in" &
      Remaining'Image &
      (if Remaining = 1 then " second" else " seconds"));
    delay 1.0;
    Remaining := Remaining - 1;
  end loop;
  Put_Line ("Boom! (goodbye Nostromo)");
```

Boolean "If Expressions"

- Return a value of either True or False
  - `(if P then Q)` - assuming **P** and **Q** are **Boolean**
  - "If P is True then the result of the *if expression* is the value of Q"
- But what is the overall result if all conditions are False?
- Answer: the default result value is True
  - Why?
    - Consistency with mathematical proving

The "else" Part When Result Is Boolean

- Redundant because the default result is True

```
(if P then Q else True)
```

- So for convenience and elegance it can be omitted

```
Acceptable : Boolean := (if P1 > 0 then P2 > 0 else True);
Acceptable : Boolean := (if P1 > 0 then P2 > 0);
```

- Use `else` if you need to return False at the end

Rationale for Parentheses Requirement

- Prevents ambiguity regarding any enclosing expression
- Problem:

```
X : Integer := if condition then A else B + 1;
```

- Does that mean
    - If condition, then **X := A + 1**, else **X := B + 1 OR**
    - If condition, then **X := A**, else **X := B + 1**
- But not required if parentheses already present
    - Because enclosing construct includes them

```
Subprogram_Call (if A then B else C);
```

When to Use If Expressions

- When you need computation to be done prior to sequence of statements

```
Shift_Differential : constant Float :=
   (if Shift = First then 1.0
    elsif Shift = Second then 1.25
    else 1.5);
```

- When an enclosing function would be either heavy or redundant with enclosing context

```
Holiday_Bonus : Float :=
   (if Hours_Worked (Week_52) >= 40 then 100.0
    elsif Hours_Worked (Week_51) >= 40 then 50.0
    else 25.0);
```

- Preconditions and postconditions

```
function Area (L, W : Float) return Float
   with
      Post => (if L < 0.0 or W < 0.0 then 0.0
               else L * W);
```

- Static named numbers

```
High_Bit_Index : constant :=
   (if Integer'Size = 32 then 31 else 63);
```

Case Expressions

- Syntax similar to *case statements*
    - Lighter: no closing **end case**
    - Commas between choices
- Same general rules as *if expressions*
    - Parentheses required unless already present
    - Type of "result" must match context
- Advantage over *if expressions* is completeness checked by compiler
- Same as with `case` statements (unless `others` is used)

```
-- compile error if not all days covered
Hours : constant Integer :=
   (case Day_of_Week is
    when Mon .. Thurs => 9,
    when Fri          => 4,
    when Sat | Sun    => 0);
```

"Case Expression" Example

```
Leap : constant Boolean :=
   (Today.Year mod 4 = 0 and Today.Year mod 100 /= 0)
   or else
   (Today.Year mod 400 = 0);
End_Of_Month : array (Months) of Days;
...
-- initialize array
for M in Months loop
  End_Of_Month (M) :=
     (case M is
      when Sep | Apr | Jun | Nov => 30,
      when Feb => (if Leap then 29 else 28),
      when others => 31);
end loop;
```

Quiz

```
function Sqrt (X : Float) return Float;
F : Float;
B : Boolean;
Z : Float := Get_Length;
```

Which statement(s) is (are) legal?

A. `F := if Z < 0.0 then Sqrt (-1.0 * Z) else Sqrt (Z);`
B. `F := Sqrt (if Z < 0.0 then -1.0 * Z else Z);`
C. `B := (if Z < 0.0 then Sqrt (-1.0 * Z) < 10.0 else True);`
D. `B := (if Z < 0.0 then Sqrt (-1.0 * Z) < 10.0);`

Quiz

```
function Sqrt (X : Float) return Float;
F : Float;
B : Boolean;
Z : Float := Get_Length;
```

Which statement(s) is (are) legal?

A. `F := if Z < 0.0 then Sqrt (-1.0 * Z) else Sqrt (Z);`
B. *`F := Sqrt (if Z < 0.0 then -1.0 * Z else Z);`*
C. *`B := (if Z < 0.0 then Sqrt (-1.0 * Z) < 10.0 else True);`*
D. *`B := (if Z < 0.0 then Sqrt (-1.0 * Z) < 10.0);`*

Explanations

A. Missing parentheses around expression
B. Legal - Expression is already enclosed in parentheses so you don't need to add more
C. Legal - **`else`** `True` not needed but is allowed
D. Legal - `B` will be `True` if Z >= 0.0

Quantified Expressions

Quantified Expressions

- Expressions that report a Boolean value about a set of objects
    - Where *set* indicates an **array** or other iterable object
- Value indicates if something is true about the set
    - Either true for **any** element in the set or for **some** element in the set
- That "something" is expressed as an arbitrary boolean expression
    - A so-called "predicate"
- *Universal quantified expression*
    - Indicates whether predicate holds for **all** components
- *Existential quantified expression*
    - Indicates 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;
```

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 ∀ (inverted 'A', for "all")
- "There is no member of the set for which the predicate does not hold"
  - If predicate is False for any member, the whole is False
- Functional equivalent

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

Universal Quantifier Illustration

- "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 ∃ (rotated 'E', for "exists")
- "There is at least one member of the set for which the predicate holds"
  - If predicate is True for any member, the whole is True
- Functional equivalent

```
function Existential (Set : Components) return Boolean is
begin
  for C of Set loop
    if Predicate (C) then
      return True; -- Need only be true for at least one
    end if;
  end loop;
  return False;
end Existential;
```

Existential Quantifier Illustration

- "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);
```

- 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) <= 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

Quiz

Which declaration(s) is (are) legal?

A.
```
function F (S : String) return Boolean is
  (for all C of S => C /= ' ');
```

B.
```
function F (S : String) return Boolean is
  (not for some C of S => C = ' ');
```

C.
```
function F (S : String) return String is
  (for all C of S => C);
```

D.
```
function F (S : String) return String is
  (if (for all C of S => C /= ' ') then "OK"
   else "NOK");
```

Quiz

Which declaration(s) is (are) legal?

A. *function F (S : String) return Boolean is (for all C of S => C /= ' ');*

```
function F (S : String) return Boolean is
  (for all C of S => C /= ' ');
```

B.
```
function F (S : String) return Boolean is
  (not for some C of S => C = ' ');
```

C.
```
function F (S : String) return String is
  (for all C of S => C);
```

D. *function F (S : String) return String is (if (for all C of S => C /= ' ') then "OK" else "NOK");*

```
function F (S : String) return String is
  (if (for all C of S => C /= ' ') then "OK"
   else "NOK");
```

B. Parentheses required around the quantified expression

C. Must return a `Boolean`

Quiz

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

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

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

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

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

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

Quiz

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

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

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

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

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

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

B. Every element of `A` must match every element of `B`. This fails when `A` and `B` contain more than one distinct value, such as: `(0, 1, 0)`

C. Returns `True` if any value in `A` matches any value in `B`, even if the arrays differ elsewhere - `A = (0, 0, 1)` and `B = (0, 1, 1)` returns `True`

Lab

Expressions Lab

- Goal
  - Use expression functions to validate array contents
- Requirements
  - Prompt has three arrays of dates
  - For each set of dates, 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

Expressions Lab Solution - Checks

```
subtype Year_T is Positive range 1_900 .. 2_099;
subtype Month_T is Positive range 1 .. 12;
subtype Day_T is Positive range 1 .. 31;

type Date_T is record
   Year  : Positive;
   Month : Positive;
   Day   : Positive;
end record;

type Dates_T is array (1 .. 3) of Date_T;

function Is_Leap_Year (Year : Positive) return Boolean
is (Year mod 400 = 0 or else (Year mod 4 = 0 and Year mod 100 /= 0));

function Days_In_Month (Month : Positive; Year : Positive) return Day_T
is (case Month is
      when 4 | 6 | 9 | 11 => 30,
      when 2 => (if Is_Leap_Year (Year) then 29 else 28),
      when others => 31);

function Is_Valid (Date : Date_T) return Boolean
is (Date.Year in Year_T
    and then Date.Month in Month_T
    and then Date.Day <= Days_In_Month (Date.Month, Date.Year));

function Any_Invalid (List : Dates_T) return Boolean
is (for some Date of List => not Is_Valid (Date));

function Same_Year (List : Dates_T) return Boolean
is (for all I in List'Range => List (I).Year = List (List'First).Year);
```

Expressions Lab Solution - Main

```
   Good_Dates : constant Dates_T :=
     ((Year => 2_025, Month => 1, Day => 2),
      (Year => 2_024, Month => 2, Day => 28),
      (Year => 2_000, Month => 2, Day => 29));

   Mixed_Dates : constant Dates_T :=
     ((Year => 2_025, Month => 4, Day => 30),
      (Year => 2_024, Month => 2, Day => 28),
      (Year => 1_900, Month => 2, Day => 29));

   Same_Year_Dates : constant Dates_T :=
     ((Year => 2_025, Month => 4, Day => 30),
      (Year => 2_025, Month => 2, Day => 28),
      (Year => 2_025, Month => 2, Day => 29));

begin

   Put_Line ("Good_Dates");
   Put_Line ("  Any invalid: " & Boolean'Image (Any_Invalid (Good_Dates)));
   Put_Line ("  Same Year: " & Boolean'Image (Same_Year (Good_Dates)));

   Put_Line ("Mixed_Dates");
   Put_Line ("  Any invalid: " & Boolean'Image (Any_Invalid (Mixed_Dates)));
   Put_Line ("  Same Year: " & Boolean'Image (Same_Year (Mixed_Dates)));

   Put_Line ("Same_Year_Dates");
   Put_Line
     ("  Any invalid: " & Boolean'Image (Any_Invalid (Same_Year_Dates)));
   Put_Line ("  Same Year: " & Boolean'Image (Same_Year (Same_Year_Dates)));
```

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

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

```ada
--  User-defined overloading
function "+" (L,R : Complex) return Complex is
begin
  return (L.Real_Part + R.Real_Part,
          L.Imaginary + R.Imaginary);
end "+";

A, B, C : Complex;
I, J, K : Integer;

I := J + K; -- overloaded operator (predefined)
A := B + C; -- overloaded operator (user-defined)
```

Benefits and Risk of Overloading

- Management of the name space
  - Support for abstraction
  - Linker will not simply take the first match and apply it globally
- Safe: compiler will reject ambiguous calls
- Sensible names are the programmer's job

```
function "+" (L, R : Integer) return String is
begin
   return Integer'Image (L - R);
end "+";
```

Enumerals and Operators

Overloading Enumerals

- Each is treated as if a function name (identifier)
- Thus same rules as for function identifier overloading

```
type Stop_Light is (Red, Yellow, Green);
type Colors is (Red, Blue, Green);
Shade : Colors := Red;
Current_Value : Stop_Light := Red;
```

Overloadable Operator Symbols

- Only those defined by the language already
  - Users cannot introduce new operator symbols
- Note that assignment (:=) is not an operator
- Operators (in precedence order)

Logicals	and, or, xor
Relationals	`<`, `<=`, `=`, `>=`, `>`
Unary	`+`, `-`
Binary	`+`, `-`, `&`
Multiplying	`*`, `/`, `mod`, `rem`
Highest precedence	`**`, `abs`, `not`

Parameters for Overloaded Operators

- Must not change syntax of calls
  - Number of parameters must remain same (unary, binary...)
  - No default expressions allowed for operators
- Infix calls use positional parameter associations
  - Left actual goes to first formal, right actual goes to second formal
  - Definition

    ```
    function "*" (Left, Right : Integer) return Integer;
    ```
  - Usage

    ```
    X := 2 * 3;
    ```
- Named parameter associations allowed but ugly
  - Requires prefix notation for call

```
X := "*" (Left => 2, Right => 3);
```

Call Resolution

Call Resolution

- Compilers must reject ambiguous calls
- *Resolution* is based on the calling context
  - Compiler attempts to find a matching **profile**
  - Based on **Parameter** and **Result** Type
- Overloading is not re-definition, or hiding
  - More than one matching profile is ambiguous

```
type Complex is ...
function "+" (L, R : Complex) return Complex;
A, B : Complex := some_value;
C : Complex := A + B;
D : Float := A + B;  -- illegal!
E : Float := 1.0 + 2.0;
```

Profile Components Used

- Significant components appear in the call itself
  - **Number** of parameters
  - **Order** of parameters
  - **Base type** of parameters
  - **Result** type (for functions)
- Insignificant components might not appear at call
  - Formal parameter **names** are optional
  - Formal parameter **modes** never appear
  - Formal parameter **subtypes** never appear
  - **Default** expressions never appear

```
Display (X);
Display (Foo => X);
Display (Foo => X, Bar => Y);
```

Manually Disambiguating Calls

- Qualification can be used
- Named parameter association can be used
  - Unless name is ambiguous

```
type Stop_Light is (Red, Yellow, Green);
type Colors is (Red, Blue, Green);
procedure Put (Light : in Stop_Light);
procedure Put (Shade : in Colors);

Put (Red);  -- ambiguous call
Put (Yellow);  -- not ambiguous: only 1 Yellow
Put (Colors'(Red)); -- using type to distinguish
Put (Light => Green); -- using profile to distinguish
```

Overloading Example

```
function "+" (Left : Position; Right : Offset)
  return Position is
begin
   return Position'(Left.Row + Right.Row, Left.Column + Right.Col);
end "+";

function Acceptable (P : Position) return Boolean;
type Positions is array (Moves range <>) of Position;

function Next (Current : Position) return Positions is
  Result : Positions (Moves range 1 .. 4);
  Count  : Moves := 0;
  Test   : Position;
begin
  for K in Offsets'Range loop
    Test := Current + Offsets (K);
    if Acceptable (Test) then
      Count := Count + 1;
      Result (Count) := Test;
    end if;
  end loop;
  return Result (1 .. Count);
end Next;
```

Quiz

```
type Vertical_T is (Top, Middle, Bottom);
type Horizontal_T is (Left, Middle, Right);
function "*" (H : Horizontal_T; V : Vertical_T) return Positive;
function "*" (V : Vertical_T; H : Horizontal_T) return Positive;
P : Positive;
```

Which statement(s) is (are) legal?

A. `P := Horizontal_T'(Middle) * Middle;`
B. `P := Top * Right;`
C. `P := "*" (Middle, Top);`
D. `P := "*" (H => Middle, V => Top);`

Quiz

```
type Vertical_T is (Top, Middle, Bottom);
type Horizontal_T is (Left, Middle, Right);
function "*" (H : Horizontal_T; V : Vertical_T) return Positive;
function "*" (V : Vertical_T; H : Horizontal_T) return Positive;
P : Positive;
```

Which statement(s) is (are) legal?

A. *`P := Horizontal_T'(Middle) * Middle;`*
B. *`P := Top * Right;`*
C. *`P := "*" (Middle, Top);`*
D. `P := "*" (H => Middle, V => Top);`

Explanations

A. Qualifying one parameter resolves ambiguity
B. No overloaded names
C. Use of `Top` resolves ambiguity
D. When overloading subprogram names, best to not just switch the order of parameters

User-Defined Equality

User-Defined Equality

- Allowed like any other operator
  - Must remain a binary operator
- Typically declared as `return Boolean`
- Hard to do correctly for composed types
  - Especially **user-defined** types
  - Issue of *Composition of equality*

Lab

Overloading Lab

- Requirements
  - Create multiple functions named `Convert` to convert between a character digits and its name
    - One routine should take a digit and return the name (e.g. **'3'** would return **three**)
    - One routine should take the name and return the digit (e.g. **two** would return **'2'**)
    - Hint: enumerals for the name will be easier than dealing with strings
  - Create overloaded addition functions that will add any combination of the two (digit and name)
    - The result can be an integer (i.e. dont worry about converting the result of `'7' + eight`)
    - Hint: It might be easier to convert one to the other before adding!
  - The prompt has four equations in the comments - use those to prove your code works

Overloading Lab Solution - Conversion Functions

```
   subtype Digit_T is Character 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 : Digit_Name_T) return Integer;

   function Convert (Value : Digit_T) return Digit_Name_T is
     (case Value is
         when '0' => Zero,  when '1' => One,   when '2' => Two,
         when '3' => Three, when '4' => Four,  when '5' => Five,
         when '6' => Six,   when '7' => Seven, when '8' => Eight,
         when '9' => Nine);

   function Convert (Value : Digit_Name_T) return Digit_T is
     (case Value is
         when Zero => '0',  when One => '1',   when Two => '2',
         when Three => '3', when Four => '4',  when Five => '5',
         when Six => '6',   when Seven => '7', when Eight => '8',
         when Nine => '9');

   function Convert (Value : Digit_Name_T) return Integer is
     (Digit_Name_T'Pos (Value));
```

Overloading Lab Solution - Operators

```
function "+" (Left  : Digit_T;
              Right : Digit_Name_T)
              return Integer;
function "+" (Left  : Digit_Name_T;
              Right : Digit_T)
              return Integer;
function "+" (Left  : Digit_T;
              Right : Digit_T)
              return Integer;
function "+" (Left  : Digit_Name_T;
              Right : Digit_Name_T)
              return Integer;

function "+" (Left  : Digit_T;
              Right : Digit_Name_T)
              return Integer is
   L : constant Digit_Name_T := Convert (Left);
begin
   return L + Right;
end "+";

function "+" (Left  : Digit_Name_T;
              Right : Digit_T)
              return Integer is
   Sum : constant Integer := Convert (Left) + Convert (Right);
begin
   return Sum;
end "+";

function "+" (Left  : Digit_T;
              Right : Digit_T)
              return Integer is
   L : constant Digit_Name_T := Convert (Left);
   R : constant Digit_Name_T := Convert (Right);
begin
   return L + R;
end "+";

function "+" (Left  : Digit_Name_T;
              Right : Digit_Name_T)
              return Integer is
  (Integer'(Convert (Left)) + Integer'(Convert (Right)));
```

Overloading Lab Solution - Main

```
begin

   --  One + 2
   Put_Line (Integer'Image (One + '2'));

   --  3 + Four
   Put_Line (Integer'Image ('3' + Four));

   --  Five + Six
   Put_Line (Integer'Image (Five + Six));

   --  7 + 8
   Put_Line (Integer'Image ('7' + '8'));
end Main;
```

Summary

Summary

- Ada allows user-defined overloading
    - Identifiers and operator symbols
- Benefits easily outweigh danger of senseless names
    - Can have nonsensical names without overloading
- Compiler rejects ambiguous calls
- Resolution is based on the calling context
    - *Parameter and Result Type Profile*
- Calling context is those items present at point of call
    - Thus modes etc. don't affect overload resolution
- User-defined equality is allowed
    - But is tricky

Packages

Introduction

Packages

- Enforce separation of user from implementation
    - In terms of compile-time visibility
    - For data
    - For type representation, when combined with `private` types
        - Abstract Data Types
- Provide basic namespace control
- Directly support software engineering principles
    - Especially in combination with `private` types
    - Modularity
    - Information Hiding (Encapsulation)
    - Abstraction
    - Separation of Concerns

Basic Syntax and Nomenclature

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

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

- Body

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

Separating Interface and Implementation

- *Implementation* and *specification* are textually distinct from each other
  - Typically in separate files
- Users can compile their code before body exists
  - All they need is the package specification
  - Users have **no** visibility over the body
  - Full user/designer 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

- Users have too much access to representation
    - Data
    - Type representation
- Changes force users to recode and retest
- Manual enforcement is not sufficient
- Why fixing bugs introduces new bugs!

Package Declarations

Package Declarations

- Required in all cases
  - Cannot have a package without the declaration
- Describe the user's interface
  - Declarations are exported to users
  - Effectively the "pin-outs" for the black-box
- When changed, requires users 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 Users

- Variables, types, exception, subprograms, etc.
  - The primary reason for separate subprogram declarations

```
package P is
   procedure This_Is_Exported;
end P;

package body P is
   procedure Not_Exported is
      ...
   procedure This_Is_Exported is
      ...
end P;
```

Referencing Other Packages

with Clause

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

Syntax

```
with_clause ::= with library_unit_name
                { , library_unit_name };
```

Example

```
with Designer; -- dependency
procedure User is
```

Referencing Exported Items

- Achieved via "dot notation"
- Package Specification

```
package Float_Stack is
  procedure Push (X : in Float);
  procedure Pop (X : out Float);
end Float_Stack;
```

- Package Reference

```
with Float_Stack;
procedure Test is
   X : Float;
begin
   Float_Stack.Pop (X);
   Float_Stack.Push (12.0);
   ...
```

with Clause Syntax

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

What To Import

- Need only name direct dependencies
  - Those actually referenced in the corresponding unit
- Will not cause compilation of referenced units
  - Unlike "include directives" of some languages

```
package A is
  type Something is ...
end A;

with A;
package B is
  type Something is record
    Component : A.Something;
  end record;
end B;

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

Bodies

Package Bodies

- Dependent on corresponding package specification
  - Obsolete if specification changed
- Users 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

```ada
package Graphics_Primitives is
  type Coordinate is digits 12;
  type Device_Coordinates is record
    X, Y : Integer;
  end record;
  type Normalized_Coordinates is record
    X, Y : Coordinate range 0.0 .. 1.0;
  end record;
  type Offset is record
    X, Y : Coordinate range -1.0 .. 1.0;
  end record;
  -- nothing to implement, so no body allowed
end Graphics_Primitives;
```

Example Spec Requiring a Package Body

```ada
package VT100 is
  subtype Rows is Integer range 1 .. 24;
  subtype Columns is Integer range 1 .. 80;
  type Position is record
    Row   : Rows := Rows'First;
    Col : Columns := Columns'First;
  end record;
   -- The following need to be defined in the body
  procedure Move_Cursor (To : in Position);
  procedure Home;
  procedure Clear_Screen;
  procedure Cursor_Up (Count : in Positive := 1);
end VT100;
```

Required Body Example

```
package body VT100 is
  -- This function is not visible outside this package
  function Unsigned (Input : Integer) return String is
    Str : constant String := Integer'Image (Input);
  begin
    return Str (2 .. Str'Length);
  end Unsigned;
  procedure Move_Cursor (To : in Position) is
  begin
    Text_IO.Put (ASCII.Esc & 'I' &
                 Unsigned (To.Row) & ';' &
                 Unsigned (To.Col) & 'H');
  end Move_Cursor;
  procedure Home is
  begin
    Text_IO.Put (ASCII.Esc & "iH");
  end Home;
  procedure Cursor_Up (Count : in Positive := 1) is ...
    ...
end VT100;
```

Quiz

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

Which completion(s) is (are) correct for `package P`?

A. `No completion is needed`

B.
```
package body P is
  procedure One (V : out Integer) is null;
end P;
```

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

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

Quiz

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

Which completion(s) is (are) correct for **package** P?

A. No completion is needed

B. *package body P is*
```
package body P is
   procedure One (V : out Integer) is null;
end P;
```

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

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

A. Procedure One must have a body
B. Parameter V is **out** but not assigned (legal but not a good idea)
C. Redeclaration of Object_One
D. Implementation of `One` is valid

Executable Parts

Optional Executable Part

```
package_body ::=
    package body name is
       declarative_part
    [ begin
       handled_sequence_of_statements ]
    end [ name ];
```

Executable Part Semantics

- Executed only once, when package is elaborated
- Ideal when statements are required for initialization
  - Otherwise initial values in variable declarations would suffice

```
package body Random is
  Seed1, Seed2 : Integer;
  Call_Count : Natural := 0;
  procedure Initialize (Seed1 : out Integer;
                        Seed2 : out Integer) is ...
  function Number return Float is ...
begin -- Random
  Initialize (Seed1, Seed2);
end Random;
```

Requiring/Rejecting Bodies Justification

- Consider the alternative: an optional package body that becomes obsolete prior to building
- Builder could silently choose not to include the package in executable
  - Package executable part might do critical initialization!

```
package P is
  Data : array (L .. U) of
      Integer;
end P;

package body P is
  ...
begin
  for K in Data'Range loop
    Data (K) := ...
  end loop;
end P;
```

Forcing a Package Body to Be Required

- Use

  `pragma Elaborate_Body`
  - Says to elaborate body immediately after spec
  - Hence there must be a body!
- Additional pragmas we will examine later

```
package P is
  pragma Elaborate_Body;
  Data : array (L .. U) of
      Integer;
end P;

package body P is
  ...
begin
  for K in Data'Range loop
    Data (K) := ...
  end loop;
end P;
```

Idioms

Named Collection of Declarations

- Exports:
  - Objects (constants and variables)
  - Types
  - Exceptions
- Does not export operations

```
package Physical_Constants is
  Polar_Radius_in_feet         : constant := 20_856_010.51;
  Equatorial_Radius_in_feet : constant := 20_926_469.20;
  Earth_Diameter_in_feet : constant := 2.0 *
     ((Polar_Radius_in_feet + Equatorial_Radius_in_feet)/2.0);
  Sea_Level_Air_Density : constant := 0.00239; --slugs/foot**3
  Altitude_Of_Tropopause_in_feet : constant := 36089.0;
  Tropopause_Temperature_in_celsius : constant := -56.5;
end Physical_Constants;
```

Named Collection of Declarations (2)

- Effectively application global data

```ada
package Equations_of_Motion is
  Longitudinal_Velocity : Float := 0.0;
  Longitudinal_Acceleration : Float := 0.0;
  Lateral_Velocity  : Float := 0.0;
  Lateral_Acceleration : Float := 0.0;
  Vertical_Velocity : Float := 0.0;
  Vertical_Acceleration : Float := 0.0;
  Pitch_Attitude : Float := 0.0;
  Pitch_Rate : Float := 0.0;
  Pitch_Acceleration : Float := 0.0;
end Equations_of_Motion;
```

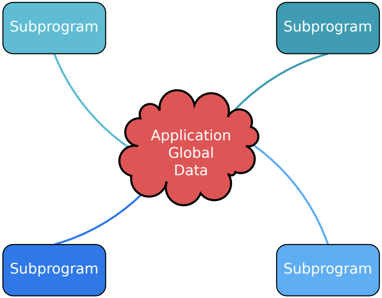
Group of Related Program Units

- Exports:
    - Objects
    - Types
    - Values
    - Operations
- Users have full access to type representations
    - This visibility may be necessary

```
package Linear_Algebra is
  type Vector is array (Positive range <>) of Float;
  function "+" (L,R : Vector) return Vector;
  function "*" (L,R : Vector) return Vector;
  ...
end Linear_Algebra;
```

Uncontrolled Data Visibility Problem

- Effects of changes are potentially pervasive so one must understand everything before changing anything

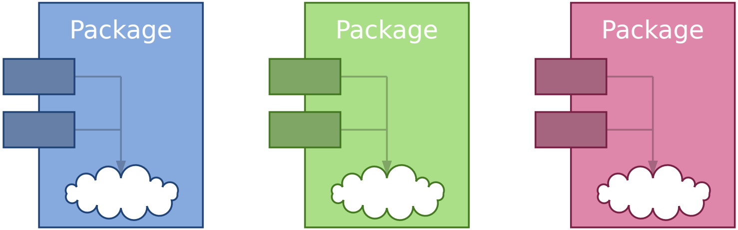


Packages and "Lifetime"

- Like a subprogram, objects declared directly in a package exist while the package is "in scope"
  - Whether the object is in the package spec or body
- Packages defined at the library level (not inside a subprogram) are always "in scope"
  - Including packages nested inside a package
- So package objects are considered "global data"
  - Putting variables in the spec exposes them to users
    - 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
  - Users can only call these visible routines
- Global change effects are much less likely
  - Direct breakage is impossible



Abstract Data Machines

- Exports:
  - Operations
  - State information queries (optional)
- No direct user access to data

```
package Float_Stack is
  Max : constant := 100;
  procedure Push (X : in Float);
  procedure Pop (X : out Float);
end Float_Stack;

package body Float_Stack is
  type Contents is array (1 .. Max) of Float;
  Values : Contents;
  Top : Integer range 0 .. Max := 0;
  procedure Push (X : in Float) is ...
  procedure Pop (X : out Float) is ...
end Float_Stack;
```

Controlling Type Representation Visibility

- In other words, support for Abstract Data Types
  - No operations visible to users based on representation
- The fundamental concept for Ada
- Requires `private` types discussed in coming section...

Lab

Packages Lab

- Requirements
  - Create a program to build a list of simple mathematical equations
  - For each equation, print out if the result would be in range
    - Equations are two real numbers and a simple operation (+, -, *, /)
- Hint: create (at least) three packages
  - **Types** creates
    - Numeric type with a range
    - Equation record type
    - Mechanism to convert the record to a string
  - **Validation**
    - Verifies equation result would be in range
  - **List** contains
    - List of equations
    - Mechanism to retrieve each item in the list
  - Remember: `with package_name;` gives access to `package_name`

Creating Packages in GNAT Studio

- Right-click on the source directory node
  - If you used a prompt, the directory is probably **.**
  - If you used the wizard, the directory is probably **src**
- New → Ada Package
  - Fill in name of Ada package
  - Check the box if you want to create the package body in addition to the package spec

Packages Lab Solution - Types

```
package Types is

   Minimum_Value : constant := 0.0;
   Maximum_Value : constant := 100.0;

   type Numeric_T is digits 6 range Minimum_Value .. Maximum_Value;

   type Record_T is record
      Left     : Numeric_T;
      Right    : Numeric_T;
      Operator : Character;
   end record;

   function Image
     (R : Record_T)
      return String;

end Types;
```

```
package body Types is

   function Image
     (R : Record_T)
      return String is
   begin
      return R.Left'Image & " " & R.Operator & " " & R.Right'Image;
   end Image;

end Types;
```

Packages Lab Solution - Validation

```
with Types;
package Validator is

   function Is_Valid
     (Object : Types.Record_T)
      return Boolean;

end Validator;
```

```
package body Validator is

   function Is_Valid
     (Object : Types.Record_T)
      return Boolean is
      Result : Float;
   begin
      case Object.Operator is
         when '+' =>
            Result := Float (Object.Left) + Float (Object.Right);
         when '-' =>
            Result := Float (Object.Left) - Float (Object.Right);
         when '*' =>
            Result := Float (Object.Left) * Float (Object.Right);
         when '/' =>
            Result := Float (Object.Left) / Float (Object.Right);
         when others =>
            --  If the operator isn't legal, make sure
            --  the result is out of range
            Result := Float (Types.Maximum_Value) * 2.0;
      end case;
      return
        Result in Float (Types.Minimum_Value) .. Float (Types.Maximum_Value);
   end Is_Valid;

end Validator;
```

Packages Lab Solution - List

```
with Types;
package List is

   procedure Add
     (Left     : Types.Numeric_T;
      Operator : Character;
      Right    : Types.Numeric_T);

   function Length return Natural;

   function Element
     (Index : Integer)
      return Types.Record_T;

end List;
```

```
package body List is

   Global : array (1 .. 100) of Types.Record_T;
   Count  : Natural := 0;

   procedure Add
     (Left     : Types.Numeric_T;
      Operator : Character;
      Right    : Types.Numeric_T) is
   begin
      Count          := Count + 1;
      Global (Count) :=
        (Left     => Left,
         Right    => Right,
         Operator => Operator);
   end Add;

   function Length return Natural is (Count);

   function Element
     (Index : Integer)
      return Types.Record_T is (Global (Index));

end List;
```

Packages Lab Solution - Main

```
with Ada.Text_IO; use Ada.Text_IO;
with List;
with Types;
with Validator;
procedure Main is

begin

   List.Add (12.34, '+', 56.78);
   List.Add (12.34, '-', 56.78);
   List.Add (12.34, '*', 56.78);
   List.Add (12.34, '/', 56.78);

   for Index in 1 .. List.Length loop
      declare
         Item : constant Types.Record_T := List.Element (Index);
      begin
         Put_Line
           (Types.Image (Item) & " " &
            Boolean'Image (Validator.Is_Valid (Item)));
      end;
   end loop;

end Main;
```

Summary

Summary

- Emphasizes separations of concerns
- Solves the global visibility problem
  - Only those items in the specification are exported
- Enforces software engineering principles
  - Information hiding
  - Abstraction
- Implementation can't be corrupted by users
  - Compiler won't let users compile references to internals
- Bugs must be in the implementation, not users
  - Only body implementation code has to be understood

Private Types

Introduction

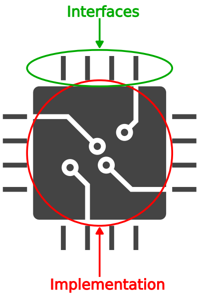
Introduction

- Why does fixing bugs introduce new ones?
- Control over visibility is a primary factor
  - Changes to an abstraction's internals shouldn't break **clients**
  - Including type representation
- Need tool-enforced rules to isolate dependencies
  - Between implementations of abstractions and their **clients**
  - In other words, "information hiding"

Information Hiding

Compare to integrated circuits

- Hides implementation details from the end **client**
- **Client** only sees the interface
  - Not how it works underneath



- Example - you can drive a car without knowing how the engine works:
  - **Interfaces:** steering wheel, pedals, etc
  - **Implementation:** engine, transmission, brake pads, etc

Views

- Specify legal manipulation for objects of a type

```
type Integer_T is range 0 .. 100;
```

  - Can use math operators, comparison operators, assignment, ...

```
type Enumerated_T is (Red, Yellow, Green);
```

  - Can use comparison operators, assignment, ...
- Some views are implicit in language

```
procedure Increment (Value  : in out Integer;
                     Amount : in     Integer);
```

  - **Value** has all operations available
  - **Amount** is read-only
- Views may be explicitly specified

```
Initial_Value : constant Float := 32.0;
```

  - **Initial_Value** cannot be assigned a new value
- **Purpose:** control usage in accordance with design
  - Adherence to interface
  - Abstract Data Types

Implementing Abstract Data Types Via Views

Implementing Abstract Data Types

- A combination of constructs in Ada
- Not based on single "class" construct, for example
- Constituent parts
  - Packages, with "private part" of package spec
  - "Private types" declared in packages
  - Subprograms declared within those packages

Package Visible and Private Parts for Views

- Declarations in visible part are exported to **clients**
- Declarations in private part are hidden from **clients**
  - 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

Syntax

```
private_type_declaration ::=
   type identifier [discriminant_part] is private;
```

- Private type declaration must occur in visible part
  - *Partial view*
  - Only partial information on the type
  - **Clients** 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 **clients**
  - **Not** visible to **designer** until reached

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

Partial and Full Views of Types

- Private type declaration defines a *partial view*
  - The type name is visible
  - Only **designer's** operations and some predefined operations
  - No references to full type representation
- Full type declaration defines the *full view*
  - Fully defined as a record type, scalar, imported type, etc...
  - Just an ordinary type within the package
- Operations available depend upon one's view

Software Engineering Principles

- Encapsulation and abstraction enforced by views
  - Compiler enforces view effects
- Same protection as hiding in a package body
  - Recall "Abstract Data Machines" idiom
- Additional flexibility of types
  - Unlimited number of objects possible
  - Passed as parameters
  - Components of array and record types
  - Dynamically allocated
  - et cetera

Clients 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 **clients** cannot compile code referencing representation

```
with Bounded_Stacks;
procedure Client is
  My_Stack : Bounded_Stacks.Stack;
begin
  My_Stack.Top := 1;  -- Client cannot see inside "Stack"
end Client;
```

```
client.adb:5:05: error: invalid prefix in selected component "My_Stack"
```

Benefits of Views

- **Clients** depend only on visible part of specification
    - Impossible for **clients** to compile references to private part
    - Physically seeing private part in source code is irrelevant
- Changes to implementation don't affect **clients**
    - No editing changes necessary for **client** code
- Implementers can create bullet-proof abstractions
    - If a facility isn't working, you know where to look
- Fixing bugs is less likely to introduce new ones

Quiz

```
package P is
   type Private_T is private;

   type Record_T is record
```

Which component(s) is (are) legal?

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

B. `Component_B : Private_T := null;`

C. `Component_C : Private_T := 0;`

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

```
    end record;
```

Quiz

```
package P is
   type Private_T is private;

   type Record_T is record
```

Which component(s) is (are) legal?

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

B. `Component_B : Private_T := null;`

C. `Component_C : Private_T := 0;`

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

```
    end record;
```

Explanations

A. Visible part does not know `Private_T` is discrete
B. Visible part does not know possible values for `Private_T`
C. Visible part does not know possible values for `Private_T`
D. Correct - type will have a known size at run-time

Private Part Construction

Private Part and Recompilation

- **Clients** 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 **clients'** code, e.g., storage layouts for private-typed objects
- Thus changes to private part require **client** 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 Visible (B : in out Private_T);
private
   -- Hidden and Hidden_T are not visible to clients
   procedure Hidden (Param : in out Private_T);
   type Hidden_T is ...;
   type Private_T is array (1 .. 3) of Hidden_T;
end Foo;

package body Foo is
   -- Local is not visible to clients
   procedure Local (Param : in out Private_T) is ...
   procedure Hidden (Param : in out Private_T) is ...
   procedure Visible (Param : in out Private_T) is ...
 end Foo;
```

Full Type Declaration

- May be any type
  - Predefined or user-defined
  - Including references to imported types
- Contents of private part are unrestricted
  - Anything a package specification may contain
  - Types, subprograms, variables, etc.

```ada
package Designer is
  type Item_T is private;
  ...
private
  type Vector is array (1.. 10)
     of Integer;
  function Initial
     return Vector;
  type Item_T record
    One, Two : Vector := Initial;
  end record;
end Designer;
```

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 Example 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 Example;
```

Quiz

```
package Example 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 Example;

package body Example is
   Object_C : Private_T;
   procedure Proc (Param : in out Private_T) is null;
end Example;
```

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

A. `Object_A`
B. `Object_B`
C. `Object_C`
D. None of the above

Quiz

```
package Example 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 Example;

package body Example is
   Object_C : Private_T;
   procedure Proc (Param : in out Private_T) is null;
end Example;
```

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

A. `Object_A`
B. *`Object_B`*
C. *`Object_C`*
D. None of the above

An object cannot be declared until its type is fully declared. `Object_A` could be declared constant, but then it would have to be finalized in the `private` section.

View Operations

View Operations

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

- **Client** 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

Clients 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;
```

Client 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
- **Client's** -declared subprograms
  - Using parameters of the exported private type
  - Dependent on **designer's** operations

Client Manipulation of Private Data

- What if a **client** needs extra visiblity?
  - "Show me the top of the stack"
- **Client** cannot see the stack directly
  - But may be able to use supplied functionality
  - Cannot reference type's representation

```
-- client 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 **client's** operation

```
package UART is
    type Instance is limited private;
    function Get_Next_Available return Instance;
[...]

declare
   A, B : UART.Instance := UART.Get_Next_Available;
begin
   if A = B -- Illegal
   then
       A := B; -- Illegal
   end if;
```

When to Use or Avoid Private Types

When to Use Private Types

- Implementation may change
  - Allows **clients** 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

```
package Valves is
   type Valve_Id_T is private;
   procedure Set (Valve : Valve_Id_T;
                  Value : Integer);
private
   type Valve_Id_T is new Integer;
end Valves;

with Valves; use Valves;
procedure Initialize is
   Hot, Cold : Valve_Id_T;
begin
   Set (Hot, Hot + Cold);
end Initialize;
```

- If `Valve_Id_T` was not private, call to **Set** would be valid
  - But doesn't make sense

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 **client** 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
```

Idioms

Effects of Hiding Type Representation

- Assume we have a database of employees
  - We want to track name, birth date, pay
- Implementation details
  - How do we store the name? Date? Pay?
  - Why should the **client** care?
- **Client** interface should be some private type and its primitives

```
package Database is
  type Employee_T is private;
  procedure Update_Name
    (Employee    : in out Employee_T;
     First, Last : String);
```

- Implementation changes do not require **client** rework
- Common idioms are a result
  - *Constructor*
  - *Selector*

Constructors

- Create **designer's** objects from **client's** values
- Usually functions

```
type Types_Pkg;
package Database is
  type Employee_T is private;
  function Make (Last_Name  : String;
                 First_Name : String;
                 Pay        : Types_Pkg.Pay_T)
                 return Employee_T;
private
  type Employee_T is record ...
end Employee_T;

with Database;
procedure Client is
   Employee : Database.Employee_T;
begin
   Employee := Database.Make
      (Last_Name  => "Flintstone",
       First_Name => "Fred",
       Pay        => 1.23);
```

Selectors

- Decompose **designer's** objects into **client's** values
- Usually functions

```
type Types_Pkg;
package Database is
  type Employee_T is private;
  function Last_Name
     (Employee : Employee_T)
      return String;
  function Pay
     (Employee : Employee_T)
      return Types_Pkg.Pay_T;

with Ada.Text_IO; use Ada.Text_IO;
with Database;
procedure Client
   (Employee : Database.Employee_T) is
begin
   Put_Line (Database.Last_Name (Employee) & ", " &
             Database.First_Name (Employee) & " => " &
             Database.Pay (Employee)'Image);
```

Lab

Private Types Lab

- Requirements
  - Implement a program to create a map such that
    - Map key is a country name
    - Map component content should include the associated continent and the colors in the flag
  - Map operations should include
    - Add a country to the map
    - Query the map for countries
    - Query each country for its content
  - Main program should
    - Print the entire map
    - Show a count of how many countries in the map are on each continent
    - Show a count of how many countries have the color red in their flag
- Hints
  - Should implement a **map** ADT (to keep track of the flags)
    - This **map** will contain the country, continent, and flag colors
  - 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**
  - `Types` package containing enumerals for continents, countries, and colors is part of the **prompt**

Private Types Lab Solution - Color_Set

```ada
with Types; use Types;
package Color_Set is

   type Color_List_T is array (Positive range <>) of Color_T;
   type Color_Set_T is private;

   Empty_Set : constant Color_Set_T;

   function Create (Colors : Color_List_T) return Color_Set_T;
   function Contains (Colors : Color_Set_T;
                      Color  : Color_T)
                      return Boolean;
   function Image (Set : Color_Set_T) return String;

private
   type Color_Set_Array_T is array (Color_T) of Boolean;
   type Color_Set_T is record
      Values : Color_Set_Array_T := (others => False);
   end record;
   Empty_Set : constant Color_Set_T :=
     (Values => (others => False));
end Color_Set;
```

```ada
package body Color_Set is
   function Create (Colors : Color_List_T) return Color_Set_T is
      Ret_Val : Color_Set_T := Empty_Set;
   begin
      for Idx in Colors'Range loop
         Ret_Val.Values (Colors (Idx)) := True;
      end loop;
      return Ret_Val;
   end Create;

   function Contains (Colors : Color_Set_T;
                      Color  : Color_T)
                      return Boolean is
      (Colors.Values (Color));

   function Image (Set   : Color_Set_T;
                   First : Color_T;
                   Last  : Color_T)
                   return String is
      Str : constant String :=
        (if Set.Values (First) then Color_T'Image (First) else "");
   begin
      if First = Last then
         return Str;
      elsif Str'Length = 0 then
         return Image (Set, Color_T'Succ (First), Last);
      else
         return Str & " " & Image (Set, Color_T'Succ (First), Last);
      end if;
   end Image;

   function Image (Set : Color_Set_T) return String is
     (Image (Set, Color_T'First, Color_T'Last));
end Color_Set;
```

Private Types Lab Solution - Countries Map (Spec)

```ada
with Types; use Types;
with Color_Set;
package Countries is
   subtype Key_T is Types.Country_T;
   type Map_Component_T is private;
   type Map_T is private;

   procedure Add
     (Map       : in out Map_T;
      Country   :        Key_T;
      Continent :        Continent_T;
      Colors    :        Color_Set.Color_Set_T);

   function Exists (Map     : Map_T;
                    Country : Key_T)
                    return Boolean;
   function Get (Map     : Map_T;
                 Country : Key_T)
                 return Map_Component_T;
   function Is_Valid (Component : Map_Component_T)
                      return Boolean;
   function Colors (Component : Map_Component_T)
                    return Color_Set.Color_Set_T;
   function Continent (Component : Map_Component_T)
                       return Types.Continent_T;
   function Country (Component : Map_Component_T)
                     return Types.Country_T;
   function Image (Item : Map_Component_T) return String;
   function Image (Map : Map_T) return String;

private

   type Map_Component_T is record
      Valid     : Boolean               := False;
      Country   : Key_T                 := Key_T'First;
      Continent : Continent_T          := Continent_T'First;
      Colors    : Color_Set.Color_Set_T := Color_Set.Empty_Set;
   end record;
   type Map_Array_T is array (1 .. 100) of Map_Component_T;
   type Map_T is record
      Values : Map_Array_T;
      Length : Natural := 0;
   end record;
end Countries;
```

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

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

procedure Add
  (Map       : in out Map_T;
   Country   :        Key_T;
   Continent :        Continent_T;
   Colors    :        Color_Set.Color_Set_T) is
   Index : constant Integer := Find (Map, Country);
begin
   if Index not in Map.Values'Range then
      declare
         New_Item : constant Map_Component_T :=
           (Country   => Country,
            Valid     => True,
            Continent => Continent,
            Colors    => Colors);
      begin
         Map.Length              := Map.Length + 1;
         Map.Values (Map.Length) := New_Item;
      end;
   end if;
end Add;

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

Private Types Lab Solution - Countries Map (Body - 2 of 2)

```
function Get
  (Map     : Map_T;
   Country : Key_T)
   return Map_Component_T is
   Index   : constant Integer := Find (Map, Country);
   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 Is_Valid (Component : Map_Component_T) return Boolean is
   (Component.Valid);
function Colors (Component : Map_Component_T)
                 return Color_Set.Color_Set_T is
   (Component.Colors);
function Continent (Component : Map_Component_T)
                    return Types.Continent_T is
   (Component.Continent);
function Country (Component : Map_Component_T)
                  return Types.Country_T is
   (Component.Country);

function Image (Item : Map_Component_T) return String is
  (Item.Country'Image & " => " & Color_Set.Image (Item.Colors));

function Image (Map : Map_T) return String is
   Ret_Val : String (1 .. 1_000);
   Next    : Integer := Ret_Val'First;
begin
   for I in 1 .. Map.Length loop
      declare
         Item : constant Map_Component_T := Map.Values (I);
         Str  : constant String          := Image (Item);
      begin
         Ret_Val (Next .. Next + Str'Length) := Image (Item) & ASCII.LF;
         Next                                := Next + Str'Length + 1;
      end;
   end loop;
   return Ret_Val (1 .. Next - 1);
end Image;
```

Private Types Lab Solution - Main

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Color_Set;
with Countries;
with Types;       use Types;
procedure Main is
   Map : Countries.Map_T;
   One : Countries.Map_Component_T;
begin

   Countries.Add (Map       => Map,
                  Country   => Types.United_States,
                  Continent => Types.North_America,
                  Colors    => Color_Set.Create
                      (Colors => (Red, White, Blue)));

   Countries.Add (Map       => Map,
                  Country   => Types.Finland,
                  Continent => Types.Europe,
                  Colors    => Color_Set.Create
                     (Colors => (Blue, White)));

   Countries.Add (Map       => Map,
                  Country   => Types.New_Zealand,
                  Continent => Types.Oceania,
                  Colors    => Color_Set.Create ((Red, White, Blue)));

   Put_Line ("=== Entire Map ===");
   Put_Line (Countries.Image (Map));

   New_Line;
   Put_Line ("=== Countries per Continent ===");
   declare
      Countries_Count : Natural;
   begin
      for Continent in Types.Continent_T'Range loop
         Countries_Count := 0;
         for Country in Types.Country_T loop
            One := Countries.Get (Map, Country);
            if Countries.Is_Valid (One)
              and then Countries.Continent (One) = Continent
            then
               Countries_Count := Countries_Count + 1;
            end if;
         end loop;
         Put_Line (Continent'Image & " => " & Countries_Count'Image);
      end loop;
   end;

   New_Line;
   Put_Line ("=== Flags with Red ===");
   declare
      Flags_With_Red : Natural := 0;
   begin
      for Country in Types.Country_T loop
         One := Countries.Get (Map, Country);
         if Countries.Is_Valid (One)
           and then Color_Set.Contains (Countries.Colors (One), Types.Red)
         then
            Flags_With_Red := Flags_With_Red + 1;
         end if;
      end loop;
      Put_Line ("Flags with red => " & Flags_With_Red'Image);
   end;

end Main;
```

Summary

Summary

- Tool-enforced support for Abstract Data Types
  - Same protection as Abstract Data Machine idiom
  - Capabilities and flexibility of types
- May also be `limited`
  - Thus additionally no assignment or predefined equality
  - More on this later
- Common interface design idioms have arisen
  - Resulting from representation independence
- Assume private types as initial design choice
  - Change is inevitable

Program Structure

Introduction

Introduction

- Moving to "bigger" issues of overall program composition
- How to compose programs out of program units
- How to define subsystems

Building a System

What Is a System?

- Also called Application or Program or ...
- Collection of *library units*
  - Which are a collection of packages or subprograms

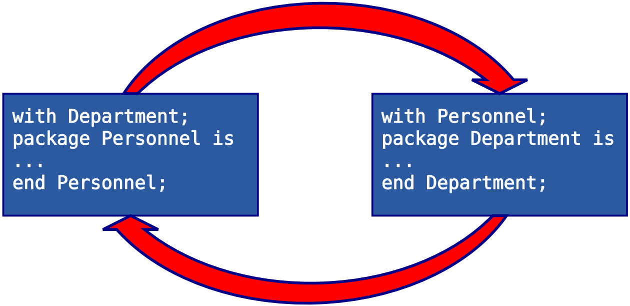
Library Units

- Those units not nested within another program unit
- Candidates
  - Subprograms
  - Packages
- Dependencies between library units via `with` clauses
  - What happens when two units need to depend on each other?

Circular Dependencies

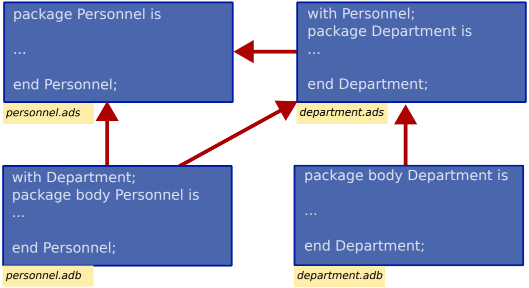
Handling Circular Dependencies

- 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;
package Personnel is
  type Employee is private;
  procedure Assign
     (This : in Employee;
      To   : in out Department.Section);
      -- We need visiblity into Department package
private
  type Employee is record
    Assigned_To : Department.Section;
  end record;
end Personnel;

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

limited with Clauses

- Solve the circular declaration dependency problem
  - Controlled circularity is 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;
```

What Is an Incomplete Type?

- 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

```
type Incomplete_T;
```

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

Legal Package Declaration Dependency

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

limited with Personnel;
package Department is
  type Section is private;
  procedure Choose_Manager (This : in out Section;
                            Who : in Personnel.Employee);
private
  type Section is record
    Manager : access Personnel.Employee;
  end record;
end Department;
```

Full **with** Clause on the Package Body

- Even though declaration has a `limited with` clause
- Typically necessary since body does the work
  - Dereferencing, etc.
- Usual semantics from then on

```
limited with Personnel;
package Department is
...
end Department;

with Personnel; -- normal view in body
package body Department is
...
end Department;
```

Hierarchical Library Units

Problem: Packages Are Not Enough

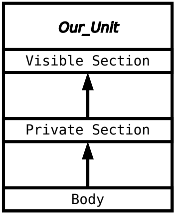
- Extensibility is a problem for private types
  - Provide excellent encapsulation and abstraction
  - But one has either complete visibility or essentially none
  - New functionality must be added to same package for sake of compile-time visibility to representation
  - Thus enhancements require editing/recompilation/retesting
- Should be something "bigger" than packages
  - Subsystems
  - Directly relating library items in one name-space
    - One big package has too many disadvantages
  - Avoiding name clashes among independently-developed code

Solution: Hierarchical Library Units

- Address extensibility issue
  - Can extend packages with visibility to parent private section
  - Extensions do not require recompilation of parent unit
  - Visibility of parent's private section 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 section** sees,
and the **private section** sees everything the **visible section** sees.



Visibility Across a Hierarchy

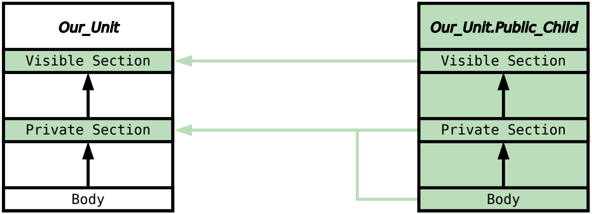
In a **package** the **body** sees everything the **private section** sees, and the **private section** sees everything the **visible section** sees.

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

Visibility Across a Hierarchy

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

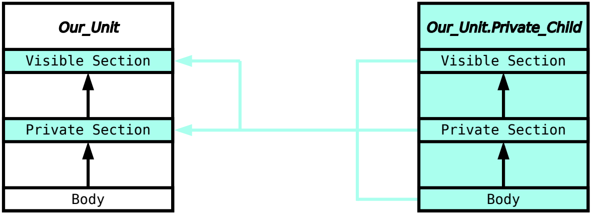
Our **child's visible section** can see our **visible section,**
and its **private section** (and **body**), can see our **private section.**



Visibility Across a Hierarchy

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

Our **private child** can see our **private section**, and **visible section**, from **anywhere**.



Programming by Extension

- *Parent unit*

```
package Complex is
  type Number is private;
  function "*" (Left, Right : Number) return Number;
  function "/" (Left, Right : Number) return Number;
  function "+" (Left, Right : Number) return Number;
  function "-" (Left, Right : Number) return Number;
...
private
  type Number is record
    Real_Part, Imaginary_Part : Float;
  end record;
end Complex;
```

- Extension created to work with parent unit

```
package Complex.Utils is
  procedure Put (C : in Number);
  function As_String (C : Number) return String;
  ...
end Complex.Utils;
```

Extension Can See Private Section

```
with Ada.Text_IO;
package body Complex.Utils is
   function As_String (Item : Number) return String is
   begin
      -- Real_Part and Imaginary_Part are
      -- visible to child's body
      return "(" &
             Item.Real_Part'Image &
             ", " &
             Item.Imaginary_Part'Image &
             ")";
   end As_String;
...
end Complex.Utils;
```

Subsystem Approach

```
with Interfaces.C;
package OS is -- Unix and/or POSIX
 type File_Descriptor is new Interfaces.C.int;
  ...
end OS;

package OS.Mem_Mgmt is
  ...
  procedure Dump (File                  : File_Descriptor;
                  Requested_Location : System.Address;
                  Requested_Size     : Interfaces.C.Size_T);
  ...
end OS.Mem_Mgmt;

package OS.Files is
  ...
  function Open (Device : Interfaces.C.char_array;
                 Permission : Permissions := S_IRWXO)
                 return File_Descriptor;
  ...
end OS.Files;
```

Predefined Hierarchies

- Standard library facilities are children of **Ada**
  - **Ada.Text_IO**
  - **Ada.Calendar**
  - **Ada.Command_Line**
  - **Ada.Exceptions**
  - et cetera
- Other root packages are also predefined
  - **Interfaces.C**
  - **Interfaces.Fortran**
  - **System.Storage_Pools**
  - **System.Storage_Elements**
  - et cetera

Hierarchical Visibility

- Children can see ancestors' visible and private sections
  - 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 ...

```
package OS is
  -- Some code
private
  type OS_Private_T is null record;
end OS;
```

```
package OS.Child is
  type Child_T is private;
private
  type Child_T is record
    Field : OS_Private_T;
  end record;
end OS.Child;
```

```
package OS.Sibling is
  type Sibling_T is private;
private
  type Sibling_T is record
    Field1 : OS_Private_T; -- OK
    Field2 : Child_T;      -- Error
  end record;
end OS.Sibling;
```

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 sections
  - Code is not in executable unless somewhere in the **with** clauses
- Explicit clauses for ancestors are redundant but OK

```
package Parent is
  ...
private
  A : Integer := 10;
end Parent;

-- no "with" of parent needed
package Parent.Child is
   ...
private
  B : Integer := Parent.A;
  -- no dot-notation needed
  C : Integer := A;
end Parent.Child;
```

with Clauses for Siblings Are Required

- If references are intended

```
with A.Foo; -- required
package body A.Bar is
   ...
   -- 'Foo' is directly visible because of the
   -- implied nesting rule
   X : Foo.Typemark;
end A.Bar;
```

Quiz

```
package Parent is
   Parent_Object : Integer;
end Parent;

package Parent.Sibling is
   Sibling_Object : Integer;
end Parent.Sibling;

package Parent.Child is
   Child_Object : Integer := ? ;
end Parent.Child;
```

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

A. `Parent.Parent_Object + Parent.Sibling.Sibling_Object`
B. `Parent_Object + Sibling.Sibling_Object`
C. `Parent_Object + Sibling_Object`
D. `None of the above`

Quiz

```
package Parent is
   Parent_Object : Integer;
end Parent;

package Parent.Sibling is
   Sibling_Object : Integer;
end Parent.Sibling;

package Parent.Child is
   Child_Object : Integer := ? ;
end Parent.Child;
```

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

A. *`Parent.Parent_Object + Parent.Sibling.Sibling_Object`*
B. *`Parent_Object + Sibling.Sibling_Object`*
C. *`Parent_Object + Sibling_Object`*
D. `None of the above`

A, B, and C are illegal because there is no reference to package `Parent.Sibling` (the reference to `Parent` is implied by the hierarchy). If `Parent.Child` had "`with Parent.Sibling;`", then A and B would be legal, but C would still be incorrect because there is no implied reference to a sibling.

Visibility Limits

Parents Do Not Know Their Children!

- Children grant themselves access to ancestors' private sections
  - May be created well after parent
  - Parent doesn't know if/when child packages will exist
- Parent body can reference children
  - Typical method of parsing out complex processes

```
with Calculator.Helper;
package Calculator is
   function Calculate (Operator    : String;
                       Left, Right : Some_T)
                       return Some_T is
   begin
      if Operator = "+" then
         return Helper.Plus (Left, Right);
```

Correlation to C++ Class Visibility Controls

- Ada private section is visible to child units

```
package P is
  A ...
private
  B ...
end P;
package body P is
  C ...
end P;
```

- Thus private section is like the protected part in C++

```
class C {
public:
  A ...
protected:
  B ...
private:
  C ...
};
```

Visibility Limits

- Visibility to parent's private section is not open-ended
  - Only visible to private sections and bodies of children
  - As if only private section of child package is nested in parent
- Recall users can only reference exported declarations
  - Child public spec only has access to parent public spec

```
package Parent is
   ...
private
   type Parent_T is ...
end Parent;

package Parent.Child is
  -- Parent_T is not visible here!
private
  -- Parent_T is visible here
end Parent.Child;

package body Parent.Child is
 -- Parent_T is visible here
end Parent.Child;
```

Children Can Break Abstraction

- Could **break** a parent's abstraction
  - Alter a parent package state
  - Alters an ADT object state
- Useful for reset, testing: fault injections...

```
package Stack is
   ...
private
   Values : array (1 .. N) of Foo;
   Top : Natural range 0 .. N := 0;
end Stack;

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

Using Children for Debug

- Provide **accessors** to parent's private information
- eg internal metrics...

```
package P is
   ...
private
  Internal_Counter : Integer := 0;
end P;
```

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

```
package body P.Child is
  function Count return Integer is
  begin
    return Internal_Counter;
  end Count;
end P.Child;
```

Quiz

```
package P is
   Object_A : Integer;
private
   Object_B : Integer;
   procedure Dummy_For_Body;
end P;

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

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

Which return statement(s) would be legal in `P.Child.X`?

A. `return Object_A;`
B. `return Object_B;`
C. `return Object_C;`
D. None of the above

Quiz

```ada
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(s) would be legal in `P.Child.X`?

A. *`return Object_A;`*
B. *`return Object_B;`*
C. `return Object_C;`
D. None of the above

Explanations

A. `Object_A` is in the public part of `P` - visible to any unit that **`with`**'s `P`
B. `Object_B` is in the private section of `P` - visible in the private section or body of any descendant of `P`
C. `Object_C` is in the body of `P`, so it is only visible in the body of `P`
D. A and B are both valid completions

Private Children

Private Children

- Intended as implementation artifacts
- Only available within subsystem
  - Rules prevent **with** clauses by clients
  - Thus cannot export anything outside subsystem
  - Thus have no parent visibility restrictions
    - Public part of child also has visibility to ancestors' private sections

```
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

- Add **private** to the **with** clause

```
private with Calculator.Helper;
```

- Public declarations can then access private siblings
  - But only in their private section
  - Still prevents exporting contents of private unit
- The specified package need not be a private unit
  - But why bother otherwise

private with Example

```
private package OS.UART is
  type Device is limited private;
  procedure Open (This : out Device;
      ...);
  ...
end OS.UART;

private with OS.UART;
package OS.Serial is
  type COM_Port is limited private;
  ...
private
  type COM_Port is limited record
    COM : OS.UART.Device;
    ...
  end record;
end OS.Serial;
```

Combining Private and Limited Withs

- Circular **limited with** clauses allowed
- A public unit can **with** a private unit
- With-ed unit only visible in the private section

```
limited with Parent.Public_Child;
private package Parent.Private_Child is
  type T is ...
end Parent.Private_Child;

limited private with Parent.Private_Child;
package Parent.Public_Child is
  ...
private
  X : access Parent.Private_Child.T;
end Parent.Public_Child;
```

Child Subprograms

- Child units can be subprograms
  - Recall syntax
  - Both public and private child subprograms
- Separate declaration required if private
  - Syntax doesn't allow `private` on subprogram bodies
- Only library packages can be parents
  - Only they have necessary scoping

```
private procedure Parent.Child;
```

Lab

Program Structure Lab

- Requirements
  - Create a message data type
    - Actual message type should be private
    - Need primitives to construct message and query contents
  - Create a child package that allows clients to modify the contents of the message
  - Main program should
    - Build a message
    - Print the contents of the message
    - Modify part of the message
    - Print the new contents of the message
- **Note: There is no prompt for this lab - you need to learn how to build the program structure**

Program Structure Lab Solution - Messages

```ada
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;
```

```ada
package body Messages is

   function Create (Kind    : Kind_T;
                    Request : Request_T;
                    Status  : Status_T)
                    return Message_T is
      (Kind => Kind, Request => Request, Status  => Status);

   function Kind (Message : Message_T) return Kind_T is
      (Message.Kind);
   function Request (Message : Message_T) return Request_T is
      (Message.Request);
   function Status (Message : Message_T) return Status_T is
      (Message.Status);

end Messages;
```

Program Structure Lab Solution - Message Modification

```
package Messages.Modify is

   procedure Kind (Message   : in out Message_T;
                   New_Value :        Kind_T);
   procedure Request (Message   : in out Message_T;
                      New_Value :        Request_T);
   procedure Status (Message   : in out Message_T;
                     New_Value :        Status_T);

end Messages.Modify;
```

```
package body Messages.Modify is

   procedure Kind (Message   : in out Message_T;
                   New_Value :        Kind_T) is
   begin
      Message.Kind := New_Value;
   end Kind;

   procedure Request (Message   : in out Message_T;
                      New_Value :        Request_T) is
   begin
      Message.Request := New_Value;
   end Request;

   procedure Status (Message   : in out Message_T;
                     New_Value :        Status_T) is
   begin
      Message.Status := New_Value;
   end Status;

end Messages.Modify;
```

Program Structure Lab Solution - Main

```
with Ada.Text_IO; use Ada.Text_IO;
with Messages;
with Messages.Modify;
procedure Main is
   Message : Messages.Message_T;
   procedure Print is
   begin
      Put_Line ("Kind => " & Messages.Kind (Message)'Image);
      Put_Line ("Request => " & Messages.Request (Message)'Image);
      Put_Line ("Status => " & Messages.Status (Message)'Image);
      New_Line;
   end Print;
begin
   Message := Messages.Create (Kind    => Messages.Command,
                               Request => 12.34,
                               Status  => 56);
   Print;
   Messages.Modify.Request (Message   => Message,
                            New_Value => 98.76);
   Print;
end Main;
```

Summary

Summary

- Hierarchical library units address important issues
  - Direct support for subsystems
  - Extension without recompilation
  - Separation of concerns with controlled sharing of visibility
- Parents should document assumptions for children
  - "These must always be in ascending order!"
- Children cannot misbehave unless imported ("with'ed")
- Not uncommon for two package specs to be interdependent
  - `limited with` can resolve circularity
  - May involve rethinking your type definitions

Visibility

Introduction

Improving Readability

- Descriptive names plus hierarchical packages makes for very long statements

```
Messages.Queue.Diagnostics.Inject_Fault (
   Fault    => Messages.Queue.Diagnostics.CRC_Failure,
   Position => Messages.Queue.Front);
```

- Operators treated as functions defeat the purpose of overloading

```
Complex1 := Complex_Types."+" (Complex2, Complex3);
```

- Ada has mechanisms to simplify hierarchies

Operators and Primitives

- *Operators*
  - Constructs which behave generally like functions but which differ syntactically or semantically
  - Typically arithmetic, comparison, and logical
- **Primitive operation**
  - Predefined operations such as = and + etc.
  - Subprograms declared in the same package as the type and which operate on the type
  - Inherited or overridden subprograms
  - For `tagged` types, class-wide subprograms
  - Enumeration literals

"use" Clauses

"use" Clauses

- **use Utilities;** provides direct visibility into public items in **Utilities**
  - *Direct Visibility* - as if object was referenced from within package being used
  - *Public Items* - any entity defined in package spec public section
- May still use expanded name

```
package Ada.Text_IO is
  procedure Put_Line (...);
  procedure New_Line (...);
  ...
end Ada.Text_IO;

with Ada.Text_IO;
procedure Hello is
  use Ada.Text_IO;
begin
  Put_Line ("Hello World");
  New_Line (3);
  Ada.Text_IO.Put_Line ("Good bye");
end Hello;
```

"use" Clause Syntax

- May have several, like **with** clauses
- Can refer to any visible package (including nested packages)

Syntax

```
use_package_clause ::= use package_name {, package_name};
```

- Can only **use** a package
  - Subprograms have no contents to **use**

"use" Clause Scope

- Applies to end of body, from first occurrence

```
package Distance_Pkg is
   Distance : Float := 12.34;
end Distance_Pkg;

package Time_Pkg is
   Time : Float := 98.76;
end Time_Pkg;

with Distance_Pkg;
with Time_Pkg;
use Distance_Pkg; -- everything in Distance_Pkg is now visible
package Speed_Pkg is
   Clicks  : Float := Distance / 1.6; -- OK
   Bad     : Float := Time / 60.0;     -- compile error
   use Time_Pkg; -- everything in Time_Pkg is now visible
   Seconds : Float := Time / 60.0;     -- OK
   function Speed return Float;
end Speed_Pkg;

package body Speed_Pkg is
  -- all of Distance_Pkg and Time_Pkg is visible here
  function Speed return Float is (Distance / Time);
end Speed_Pkg;
```

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 Distance_Pkg is
  Distance : Float;
end Distance_Pkg;

with Distance_Pkg;
procedure Example is
  Distance, Miles : Float;
begin
  declare
    use Distance_Pkg;
  begin
    -- With or without the clause, "Distance" means Example.Distance
    Miles := Distance;
  end;
end Example;
```

No Ambiguity

```
package Miles_Pkg is
  Distance : Float;
end Miles_Pkg;

package Kilometers_Pkg is
  Distance : Float;
end Kilometers_Pkg;

with Miles_Pkg, Kilometers_Pkg;
procedure Example is
   use Miles_Pkg, Kilometers_Pkg;
   Miles      : Float;
   Kilometers : Float;
begin
   Miles := Distance;                        -- compile error
   Kilometers := Kilometers_Pkg.Distance; -- OK
end Example;
```

"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 Types_Pkg is
  type Int is range Lower .. Upper;
  -- implicit declarations
  -- function "+"(Left, Right : Int) return Int;
  -- function "="(Left, Right : Int) return Boolean;
end Types_Pkg;

with Types_Pkg;
procedure Test is
  A, B, C : Types_Pkg.Int := some_value;
begin
  C := A + B; -- compile error (cannot see operator)
  C := Types_Pkg."+" (A,B);
  declare
    use Types_Pkg;
  begin
    C := A + B; -- operator now visible
  end;
end Test;
```

"use type" and "use all type" Clauses

"use type" and "use all type"

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

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

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

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

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

Example Code

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

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

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

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

end Types;
```

"use" Clauses Comparison

Blue = context clause being used

Red = compile errors with the context clause

No "use" clause

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

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

end Example;
```

"use" clause

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

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

end Example;
```

"use type" clause

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

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

end Example;
```

"use all type" clause

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

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

end Example;
```

Multiple "use type" Clauses

- May be necessary
- Only those that mention the type in their profile are made visible

```
package Types_Pkg 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 Types_Pkg;

with Types_Pkg;
procedure Use_Type is
  X1 : Types_Pkg.T1;
  X2 : Types_Pkg.T2;
  X3 : Types_Pkg.T3;
  use type Types_Pkg.T1;
begin
  X3 := X1 + X2; -- operator visible because it uses T1
  X2 := X2 + X2; -- operator not visible
end Use_Type;
```

Renaming Entities

Three Positives Make a Negative

- Good Coding Practices ...
  - Descriptive names
  - Modularization
  - Subsystem hierarchies
- Can result in cumbersome references

```
-- use cosine rule to determine distance between two points,
-- given angle and distances between observer and 2 points
-- A**2 = B**2 + C**2 - 2*B*C*cos(angle)
Observation.Sides (Viewpoint_Types.Point1_Point2) :=
  Math_Utilities.Square_Root
    (Observation.Sides (Viewpoint_Types.Observer_Point1)**2 +
     Observation.Sides (Viewpoint_Types.Observer_Point2)**2 -
     2.0 * Observation.Sides (Viewpoint_Types.Observer_Point1) *
       Observation.Sides (Viewpoint_Types.Observer_Point2) *
       Math_Utilities.Trigonometry.Cosine
         (Observation.Vertices (Viewpoint_Types.Observer)));
```

Writing Readable Code - Part 1

- We could use **use** on package names to remove some dot-notation

```
-- use cosine rule to determine distance between two points, given angle
-- and distances between observer and 2 points A**2 = B**2 + C**2 -
-- 2*B*C*cos(angle)
Observation.Sides (Point1_Point2) :=
  Square_Root
    (Observation.Sides (Observer_Point1)**2 +
     Observation.Sides (Observer_Point2)**2 -
     2.0 * Observation.Sides (Observer_Point1) *
       Observation.Sides (Observer_Point2) *
       Cosine (Observation.Vertices (Observer)));
```

- But that only shortens the problem, not simplifies it
  - If there are multiple "use" clauses in scope:
    - Reviewer may have hard time finding the correct definition
    - Homographs may cause ambiguous reference errors
- We want the ability to refer to certain entities by another name (like an alias) with full read/write access (unlike temporary variables)

The "renames" Keyword

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

```
package Trig renames Math.Trigonometry
```

  - Objects (or components of objects)

```
Angles : Viewpoint_Types.Vertices_Array_T
         renames Observation.Vertices;
Required_Angle : Viewpoint_Types.Vertices_T
         renames Viewpoint_Types.Observer;
```

  - Subprograms

```
function Sqrt (X : Base_Types.Float_T)
               return Base_Types.Float_T
               renames Math.Square_Root;
```

Writing Readable Code - Part 2

- With **renames** our complicated code example is easier to understand
  - Executable code is very close to the specification
  - Declarations as "glue" to the implementation details

```
begin
   package Math renames Math_Utilities;
   package Trig renames Math.Trigonometry;

   function Sqrt (X : Base_Types.Float_T) return Base_Types.Float_T
     renames Math.Square_Root;
   function Cos ...

   B : Base_Types.Float_T
     renames Observation.Sides (Viewpoint_Types.Observer_Point1);
   -- Rename the others as Side2, Angles, Required_Angle, Desired_Side
begin
   ...
   -- A**2 = B**2 + C**2 - 2*B*C*cos(angle)
   A := Sqrt (B**2 + C**2 - 2.0 * B * C * Cos (Angle));
end;
```

Renames in Ada 2022

Ada 2022

- Ada 2022 allows simpler renames for objects
  - If you are renaming an object, don't you already know the type?

```
type Array_T is array (1 .. 10) of Integer;
Global : Array_T;

begin
  for Index in Global'First .. Global'Last loop
     declare
        Ada2012 : Integer renames Global(Index);
        Ada2022 : renames Global (Index);
```

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!

Visibility Lab Solution - Types

```
package Quads is

   Number_Of_Sides : constant Natural := 4;
   type Side_T is range 0 .. 1_000;
   type Shape_T is array (1 .. Number_Of_Sides) of Side_T;

end Quads;
```

```
package Triangles is

   Number_Of_Sides : constant Natural := 3;
   type Side_T is range 0 .. 1_000;
   type Shape_T is array (1 .. Number_Of_Sides) of Side_T;

end Triangles;
```

Visibility Lab Solution - Main #1

```
with Ada.Text_IO; use Ada.Text_IO;
with Quads;
with Triangles;
procedure Main1 is

   use type Quads.Side_T;
   Q_Sides    : Natural renames Quads.Number_Of_Sides;
   Quad       : constant Quads.Shape_T := (1, 2, 3, 4);
   Quad_Total : Quads.Side_T := 0;

   use type Triangles.Side_T;
   T_Sides        : Natural renames Triangles.Number_Of_Sides;
   Triangle       : constant Triangles.Shape_T := (1, 2, 3);
   Triangle_Total : Triangles.Side_T := 0;

begin

   for I in 1 .. Q_Sides loop
      Quad_Total := Quad_Total + Quad (I);
   end loop;
   Put_Line ("Quad: " & Quads.Side_T'Image (Quad_Total));

   for I in 1 .. T_Sides loop
      Triangle_Total := Triangle_Total + Triangle (I);
   end loop;
   Put_Line ("Triangle: " & Triangles.Side_T'Image (Triangle_Total));

end Main1;
```

Visibility Lab Solution - Main #2

```
with Ada.Text_IO; use Ada.Text_IO;
with Quads;       use Quads;
with Triangles;   use Triangles;
procedure Main2 is
   function Q_Image (S : Quads.Side_T) return String
      renames Quads.Side_T'Image;
   Quad       : constant Quads.Shape_T := (1, 2, 3, 4);
   Quad_Total : Quads.Side_T := 0;

   function T_Image (S : Triangles.Side_T) return String
      renames Triangles.Side_T'Image;
   Triangle       : constant Triangles.Shape_T := (1, 2, 3);
   Triangle_Total : Triangles.Side_T := 0;

begin

   for I in Quad'Range loop
      Quad_Total := Quad_Total + Quad (I);
   end loop;
   Put_Line ("Quad: " & Q_Image (Quad_Total));

   for I in Triangle'Range loop
      Triangle_Total := Triangle_Total + Triangle (I);
   end loop;
   Put_Line ("Triangle: " & T_Image (Triangle_Total));

end Main2;
```

Summary

Summary

- `use` clauses are not evil but can be abused
  - Can make it difficult for others to understand code
- `use all type` clauses are more likely in practice than `use type` clauses
  - Added benefit: if the type being used is an enumerated type, all enumerals become visible as well
- `Renames` allow us to alias entities to make code easier to read
  - Subprogram renaming has many other uses, such as adding / removing default parameter values

Access Types

Introduction

Access Types Design

- A memory-addressed object is called an *access type*
- Objects are associated with *pools* of memory
  - Different allocation / deallocation policies
  - Each access type is unique - no conversion possible
- Access objects are **guaranteed** to always be meaningful
  - So long as `Unchecked_Deallocation` is not used
  - And when tied to a specific memory pool

Each access type defines its own safe memory domain, managed by its pool.

Access Types Can Be Dangerous

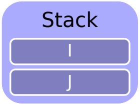
- Multiple memory issues
  - Leaks / corruptions
- Introduce potential random failures complicated to analyze
- Increase the complexity of the data structures
- May decrease the performance of the application
  - Dereferences are slightly more expensive than direct access
  - Allocations are a lot more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

Tip

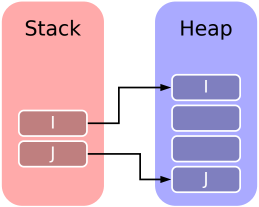
Only use them when needed

Stack Vs Heap

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




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



Access Types

Access Type

- An access type is similar to most other types
    - `access` indicates what the access type points to

```
type Rec_T is null record;
type Rec_Access_T is access Rec_T;
Rec_Ptr : Rec_Access_T;
```

- Conversion is **not** possible between this kind of access type

```
type Rec_Access_2 is access Rec_T;
Rec_Ptr_2 : Rec_Access_2 := Rec_Access_2 (Rec_Ptr);
```

example.adb:6:32: error: target type must be general access type

Note

A *general access type* is special kind of access type not handled in this course. The error message is indicating only those kinds of access types may be converted.

Allocations

- Objects are created with the **new** reserved word

```
Rec_Ptr := new Rec_T;
```

- The created object must be constrained
  - The constraint is given during the allocation

```
type String_Access_T is access String;
String_Ptr_1 : String_Access_T := new String (1..10);
```

- The object can also be created by copying an existing object
  - Using a type qualifier

```
String_Ptr_2 : String_Access_T  := new String'("abc");
Integer_Ptr  : Integer_Access_T := new Integer'(123);
```

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)
  - Works on an object, reset to `null` afterwards

Deallocation Example

```
-- generic used to deallocate memory
with Ada.Unchecked_Deallocation;
procedure Proc is
   type Object_T is null record;
   type Access_T is access Object_T;
   -- create instances of deallocation function
   procedure Free is new Ada.Unchecked_Deallocation
     (Object_T, Access_T);
   Ptr : Access_T := new Object_T;
begin
   Free (Ptr);
   -- Ptr is now null
end Proc;
```

Access Type Usage

Null Values

- 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 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

- Subprograms using an access type are **primitives of the access type**
  - **Not** the type of the accessed object

```
type Rec_T is null record;
type Rec_Access_T is access Rec_T;
procedure Proc1 (Param : Rec_T);          -- primitive of Rec_T
procedure Proc2 (Param : Rec_Access_T);  -- primitive of Rec_Access_T
```

Dereferencing Access Types

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

Dereference Examples

```
type Rec_T is record
  Field : Integer;
end record;
type Integer_Acc is access Integer;
type String_Acc is access String;
type Rec_Acc is access Rec_T;

Integer_Ptr : Integer_Acc := new Integer;
String_Ptr  : String_Acc  := new String'("abc");
Rec_Ptr     : Rec_Acc  := new Rec_T;

-- Legal
Integer_Ptr.all  := 0;
String_Ptr.all   := "cde";
String_Ptr(1)    := 'z';  -- or String_Ptr.all(1)
Rec_Ptr.all      := (Field => 987);
Rec_Ptr.Field    := 123;  -- or Rec_Ptr.all.Field

-- Compile Errors
Integer_Ptr := 0;
String_Ptr  := "cde";
Rec_Ptr     := (Field => 987);
```

Memory Corruption

Dealing with Access Types

- Access types introduce many issues
- Access types point to a location in memory
    - Modifying the pointer can point to bad locations
    - Clearing the pointer can lead to excessive memory issues
    - And lots more
- These issues are not language-specific!

Uninitialized Pointers

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

- **Ada:** this is a problem because access type objects are initialized to null
- **Other Languages:** no guarantee that the pointer is null, so you might write to a random memory location

Freeing Already-Freed Memory

```
declare
   type An_Access is access Integer;
   procedure Free is new
      Ada.Unchecked_Deallocation (Integer, An_Access);
   Object_1 : An_Access := new Integer;
   Object_2 : An_Access := Object_1;
begin
   Free (Object_1);
   delay 1.0;
   Free (Object_2);
```

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

Referencing Already-Freed Memory

```
declare
   type An_Access is access Integer;
   procedure Free is new
      Ada.Unchecked_Deallocation (Integer, An_Access);
   Object_1 : An_Access := new Integer;
   Object_2 : An_Access := Object_1;
begin
   Free (Object_1);
   Object_2.all := 5;
```

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

Memory Leak

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

Lab

Access Types Lab

- Build an application that adds / removes items from a linked list
  - At any time, user should be able to
    - Add a new item into the "appropriate" location in the list
    - Remove an item without changing the position of any other item in the list
    - Print the list
- Required goals
  1. Implement **Add** functionality
     - For this step, "appropriate" means either end of the list (but consistent - always front or always back)
  2. Implement **Print** functionality
  3. Implement **Delete** functionality

Lab Solution - Database

```ada
package Database is
   type Database_T is private;

   function Create (Number : Positive;
                    Symbol : Character)
                    return Database_T;

   function Image (Value : Database_T) return String;

   function "<" (Left, Right : Database_T) return Boolean;

private
   type Database_T is record
      Number : Positive;
      Symbol : Character;
   end record;
end Database;
```

```ada
package body Database is

   function Create (Number : Positive;
                    Symbol : Character)
                    return Database_T is
      Retval : constant Database_T :=
        (Number => Number,
         Symbol => Symbol);
   begin
      return Retval;
   end Create;

   function Image (Value : Database_T) return String is
   begin
      return Value.Symbol & Positive'Image (Value.Number);
   end Image;

   function "<" (Left, Right : Database_T) return Boolean is
   begin
      if Left.Symbol < Right.Symbol then
         return True;
      elsif Left.Symbol > Right.Symbol then
         return False;
      else
         return Left.Number < Right.Number;
      end if;
   end "<";

end Database;
```

Lab Solution - Database_List (Spec)

```
with Database; use Database;
package Database_List is
   type List_T is limited private;
   procedure First (List : in out List_T);
   procedure Next (List : in out List_T);
   function End_Of_List (List : List_T) return Boolean;
   function Current (List : List_T) return Database_T;
   procedure Insert (List      : in out List_T;
                     Component :        Database_T);
   procedure Delete (List      : in out List_T;
                     Component :        Database_T);
   function Is_Empty (List : List_T) return Boolean;
private
   type Linked_List_T;
   type Linked_List_Ptr_T is access all Linked_List_T;
   type Linked_List_T is record
      Next    : Linked_List_Ptr_T;
      Content : Database_T;
   end record;
   type List_T is record
      Head    : Linked_List_Ptr_T;
      Current : Linked_List_Ptr_T;
   end record;
end Database_List;
```

Lab Solution - Database_List (Helper Objects)

```
function Is_Empty (List : List_T) return Boolean is
begin
   return List.Head = null;
end Is_Empty;

procedure First (List : in out List_T) is
begin
   List.Current := List.Head;
end First;

procedure Next (List : in out List_T) is
begin
   if not Is_Empty (List) then
      if List.Current /= null then
         List.Current := List.Current.Next;
      end if;
   end if;
end Next;

function End_Of_List (List : List_T) return Boolean is
begin
   return List.Current = null;
end End_Of_List;

function Current (List : List_T) return Database_T is
begin
   return List.Current.Content;
end Current;
```

Lab Solution - Database_List (Insert/Delete)

```
   procedure Insert (List      : in out List_T;
                     Component :        Database_T) is
      New_Component : constant Linked_List_Ptr_T := new Linked_List_T'
          (Next    => null,
           Content => Component);
   begin
      if Is_Empty (List) then
         List.Head    := New_Component;
         List.Current := List.Head;
      else
         New_Component.Next := List.Head;
         List.Head          := New_Component;
         List.Current       := List.Head;
      end if;
   end Insert;

   procedure Free is new Ada.Unchecked_Deallocation
     (Linked_List_T, Linked_List_Ptr_T);

   procedure Delete
     (List      : in out List_T;
      Component :        Database_T) is
      To_Delete : Linked_List_Ptr_T := null;
   begin
      if not Is_Empty (List) then
         if List.Head.Content = Component then
            To_Delete    := List.Head;
            List.Head    := List.Head.Next;
            List.Current := List.Head;
         else
            declare
               Previous : Linked_List_Ptr_T := List.Head;
               Current  : Linked_List_Ptr_T          := List.Head.Next;
            begin
               while Current /= null loop
                  if Current.Content = Component then
                     To_Delete     := Current;
                     Previous.Next := Current.Next;
                     exit;
                  end if;
                  Previous := Current;
                  Current  := Current.Next;
               end loop;
            end;
            List.Current := List.Head;
         end if;
         if To_Delete /= null then
            Free (To_Delete);
         end if;
      end if;
   end Delete;

end Database_List;
```

Lab Solution - Main

```
with Ada.Text_IO; use Ada.Text_IO;
with Database;
with Database_List;
procedure Main is
   List      : Database_List.List_T;
   Component : Database.Database_T;

   procedure Add (Number : Positive;
                  Symbol : Character) is
   begin
      Database_List.Insert (List, Database.Create (Number, Symbol));
   end Add;

   procedure Delete (Number : Positive;
                     Symbol : Character) is
   begin
      Database_List.Delete (List, Database.Create (Number, Symbol));
   end Delete;

   procedure Print is
   begin
      Database_List.First (List);
      Put_Line ("List");
      while not Database_List.End_Of_List (List) loop
         Component := Database_List.Current (List);
         Put_Line ("  " & Database.Image (Component));
         Database_List.Next (List);
      end loop;
   end Print;

begin

   Add (1, 'Z');
   Add (2, 'A');
   Add (3, 'Y');
   Add (4, 'B');
   Print;
   Delete (2, 'A');
   Add (5, 'M');
   Print;

end Main;
```

Summary

Summary

- Access types are very similar to C/C++ pointers
  - Pointing to some memory location
  - Deallocation causes problems
- But Ada does a lot to remove the **need** for access types
  - 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;
```

Solution: Generics

- A *generic unit* is a unit that does not exist as part of the application
  - It is a blueprint that can work with different types, values, and even other subprograms
  - The generic uses placeholders (*generic formal data*) instead of actual entities
- The programmer creates an *instance* of the generic by specifying the actual entities for the placeholders
  - Also referred to as an *instantiation*

Ada Generic Compared to C++ Template

Ada Generic

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

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

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

C++ Template

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

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

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

Creating Generics

Declaration

- Subprograms

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

- Packages

```
generic
   type T is private;
package Stack is
   procedure Push (Item : T);
end Stack;
```

- Body is required
  - Will be specialized and compiled for **each instance**
- Children of generic units have to be generic themselves

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

Usage

- Instantiated with the new keyword

```
--  Standard library
function Convert is new Ada.Unchecked_Conversion
  (Integer, Array_Of_4_Bytes);
--  Callbacks
procedure Parse_Tree is new Tree_Parser
  (Visitor_Procedure);
--  Containers, generic data-structures
package Integer_Stack is new Stack (Integer);
```

- Advanced usages for testing, proof, meta-programming

Quiz

Which one(s) of the following can be made generic?

```
generic
   type T is private;
<answer>
```

A. `package`
B. `record`
C. `function`
D. `array`

Quiz

Which one(s) of the following can be made generic?

```
generic
   type T is private;
<answer>
```

A. *package*
B. record
C. *function*
D. array

Only packages, functions, and procedures, can be made generic.

Generic Data

Generic Types Parameters (1/3)

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

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

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

Generic Types Parameters (2/3)

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

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

Generic Types Parameters (3/3)

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

```
generic
  type Any_T (<>) is private;
procedure Generic_Procedure (V : Any_T);
```

```
1 procedure Generic_Procedure (V : Any_T) is
2   Good : Any_T := V; -- OK, can constrain by initialization
3   Bad  : Any_T;      -- Compilation error, no constraint to this
4 begin
```

```
generic_procedure.adb:3:11: error: unconstrained subtype not allowed (need initialization)
```

  - Instantiations

```
4 type Limited_T is limited null record;
5
6 -- unconstrained types are accepted
7 procedure Unconstrained is new Generic_Procedure (String);
8
9 -- type is already constrained
10 -- (but generic will still always initialize objects)
11 procedure Constrained is new Generic_Procedure (Integer);
12
13 -- Illegal: the type can't be limited because the generic
14 -- thinks it can make copies
15 procedure Bad is new Generic_Procedure (Limited_T);
```

```
instances.ads:15:44: error: actual for non-limited "Any_T" cannot be a limited type
```

Generic Parameters Can Be Combined

- Consistency is checked at compile-time

```
generic
   type T (<>) is private;
   type Acc is access 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 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);
```

Quiz

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

Which is (are) legal instantiation(s)?

A. `procedure A is new G (String, Character);`
B. `procedure B is new G (Character, Integer);`
C. `procedure C is new G (Integer, Boolean);`
D. `procedure D is new G (Boolean, String);`

Quiz

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

Which is (are) legal instantiation(s)?

A. `procedure A is new G (String, Character);`
B. *`procedure B is new G (Character, Integer);`*
C. *`procedure C is new G (Integer, Boolean);`*
D. *`procedure D is new G (Boolean, String);`*

T1 must be discrete - so an integer or an enumeration. T2 can be any type

Generic Formal Data

Generic Constants/Variables As Parameters

- Variables can be specified on the generic contract
- The mode specifies the way the variable can be used:
  - **in** → read only
  - **in out** → read write
- Generic variables can be defined after generic types

- Generic package

```
generic
  type 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);
```

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?

A. `procedure Write_A is new Write (Integer, Numeric)`
B. `procedure Write_B is new Write (Boolean, Enumerated)`
C. `procedure Write_C is new Write (Integer, 1234)`
D. `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?

A. *`procedure Write_A is new Write (Integer, Numeric)`*
B. *`procedure Write_B is new Write (Boolean, Enumerated)`*
C. `procedure Write_C is new Write (Integer, 1234)`
D. `procedure Write_D is new Write (Float, Floating_Point)`

A. `Integer` matches restrictions of `Component_T` and `Numeric` is the appropriate type
B. `Boolean` matches restrictions of `Component_T` and `Enumerated` is the appropriate type
C. The second generic parameter has to be a variable
D. The first generic parameter has to be discrete

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 - Default Values (1/2)

```
generic
   type Type_T is private;
   with function "*" (L, R : Type_T) return Type_T is <>;
function Calculate (L, W : Type_T) return Type_T;
```

- **is** <>
  - If no subprogram specified for instance, compiler uses subprogram with **same**:
    - Name
    - Parameter profile (types only, not parameter name)
- Instantiations

```
type Record_T is record
   Field : Integer;
end record;
function Multiply (L, R : Record_T) return Record_T;

function Allow_Default is new Calculate (Integer);
function Specify_Operator is new Calculate (Record_T, Multiply);
function Need_Operator is new Calculate (Record_T);

* :ada:`Allow_Default` uses the implicit definition for :ada:`*`
* :ada:`Specify_Operator` passes in the appropriate definition via :ada:`Multiply`
* :ada:`Need_Operator` generates a compile error

:error:`main.adb:11:4: error: instantiation error at gen.ads:5`

:error:`gen.ads:5:1: error: instantiation error at gen.ads:5`

:error:`main.adb:11:4: error: no visible subprogram matches the specification for "*"`
```

Generic Subprogram Parameters - Default Values (2/2)

```
2  procedure Flip (Switch : in out Boolean);
3
4  generic
5    with procedure Toggle (Switch : in out Boolean) is null;
6  procedure Print (Switch : in out Boolean);
7
8  procedure Print (Switch : in out Boolean) is
9  begin
10    Toggle (Switch);
11    Put_Line (Switch'Image);
12 end Print;
```

- `is null` (for procedures only)
  - If no procedure is specified, a null procedure will be used
- `procedure Instance1 is new Print;`
  - Line 10 will call a null subprogram because generic formal parameter `Toggle` is not specified, so line 5 forces it to be a null subprogram
- `procedure Instance2 is new Print (Flip);`
  - Line 10 will call the implementation of the procedure defined on line 2, because that procedure is passed as the generic formal parameter `Toggle`

Generic Completion

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 Formal_T is private;
package Generic_Package is
   Pointer : access Formal_T;
end Generic_Package;
```

```
1 with Generic_Package;
2 package Example is
3    type Actual_T is private;
4    package Instance is new Generic_Package (Actual_T);
5 private
6    type Actual_T is null record;
7 end Example;
```

```
example.ads:4:45: error: premature use of private type
```

Generic Incomplete Parameters

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

```
generic
   type Formal_T; -- incomplete
package Generic_Package is
   Pointer : access Formal_T;
end Generic_Package;

package Example is
   type Actual_T is private;
   package Instance is new Generic_Package (Actual_T);
private
   type Actual_T is null record;
end Example;
```

Quiz

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

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

A. `Flag := A1 /= null`
B. `Flag := A1.all'Size > 32`
C. `Flag := A2 = B2`
D. `Flag := A2 - B2 /= 0`

Quiz

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

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

A. *`Flag := A1 /= null`*
B. `Flag := A1.all'Size > 32`
C. *`Flag := A2 = B2`*
D. `Flag := A2 - B2 /= 0`

A. Can always check an access for `null`
B. T1 is incomplete, so we don't know its size
C. Comparison of private types is allowed
D. We do not know if T2 allows math

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

Genericity Lab Solution - Generic (Spec)

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

   type List_T is private;

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

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

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

Genericity Lab Solution - Generic (Body)

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

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

   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);
               This.Values (J + 1) := Temp;
            end if;
         end loop;
      end loop;
   end Sort;

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

end Generic_List;
```

Genericity Lab Solution - Main

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

   My_List : List.List_T;

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

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

Summary

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
- Generics aren't always the best solution
  - Sometimes a common routine will work just as well

Tagged Derivation

Introduction

Object-Oriented Programming with Tagged Types

- Only for **record** types

```
type Root_T is tagged record
...
```

- Child types can add new components
- Types can be **extended** by other packages
  - Conversion and qualification to base type is allowed

Tagged Derivation Ada Vs C++

```ada
type Cycle_T is tagged record
  Number_Wheels : Positive;
end record;

function Wheels
   (This : Cycle_T)
    return Positive;

type Unicycle_T is new Cycle_T
with record
  Seat_Height : Float;
end record;

overriding
function Wheels
   (This : Unicycle_T)
    return Positive is (1);
function Seat
   (This : Unicycle_T)
    return Float;
```

```cpp
class Cycle {
  public:
    unsigned NumberWheels;
    virtual unsigned Wheels(void);
  };

class Unicycle : public Cycle {
  public:
    float SeatHeight;
    virtual unsigned Wheels(void);
    virtual float Seat(void);
  };
```

Tagged Derivation

Difference with Simple Derivation

- Tagged derivation **can** change the structure of a type
  - Keywords `tagged record` and `with record`

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

type Child_T is new Root_T with record
   F2 : Integer;
end record;

Root_Object  : Root_T := (F1 => 101);
Child_Object : Child_T := (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_T is new Root_T with null record;
type Child_T is new Root_T; -- illegal
```

- Conversion is only allowed from **child to parent**

```
Root : Root_T;
Child : Child_T;
...
Root := Root_T (Child);
Child := Child_T (Root); -- 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_T is tagged null record;
type Root2_T is tagged null record;

procedure P1 (Root1 : Root1_T;
              Root2 : Root1_T);
procedure P2 (Root1 : Root1_T;
              Root2 : Root2_T); -- illegal
```

Freeze Point for Tagged Types

- Freeze point definition does not change
  - A variable of the type is declared
  - The type is derived
  - The end of the scope is reached
- Declaring tagged type primitives past freeze point is **forbidden**

```ada
type Root_T is tagged null record;
procedure Prim (Root : Root_T);

type Child_T is new Root_T with null record; -- freeze root

procedure Prim2 (Root : Root_T); -- compile error

Child : Child_T; -- freeze child

procedure Prim3 (Child : Child_T); -- compile error
```

```
example.ads:6:04: warning: no primitive operations for "Root_T" after this line
```

```
example.ads:8:14: error: this primitive operation is declared too late
```

```
example.ads:10:04: warning: no primitive operations for "Child_T" after this line
```

```
example.ads:12:14: error: this primitive operation is declared too late
```

Tagged Aggregate

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

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

type Child_T is new Root_T with record
    F2 : Integer;
end record;

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

"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
```

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*
Object.Prim1;

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

Note

This is also called *distinguished receiver* notation.

Quiz

Which declaration(s) will make P a primitive of T1?

A.
```
type T1 is tagged null record;
procedure P (O : T1) is null;
```

B.
```
type T0 is tagged null record;
type T1 is new T0 with null record;
type T2 is new T0 with null record;
procedure P (O : T1) is null;
```

C.
```
type T1 is tagged null record;
Object : T1;
procedure P (O : T1) is null;
```

D.
```
package Nested is
  type T1 is tagged null record;
end Nested;
use Nested;
procedure P (O : T1) is null;
```

Quiz

Which declaration(s) will make `P` a primitive of `T1`?

A.
```
type T1 is tagged null record;
procedure P (O : T1) is null;
```

B.
```
type T0 is tagged null record;
type T1 is new T0 with null record;
type T2 is new T0 with null record;
procedure P (O : T1) is null;
```

C.
```
type T1 is tagged null record;
Object : T1;
procedure P (O : T1) is null;
```

D.
```
package Nested is
  type T1 is tagged null record;
end Nested;
use Nested;
procedure P (O : T1) is null;
```

A. Primitive (same scope)
B. Primitive (T1 is not yet frozen)
C. T1 is frozen by the object declaration
D. Primitive must be declared in same scope as type

Quiz

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;
```

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;
```

Explanations

A. Cannot extend a non-tagged type

B. Correct

C. Components must have distinct names

D. Types derived from a tagged type must have an extension

Lab

Tagged Derivation Lab

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

Tagged Derivation Lab Solution - Types (Spec)

```ada
package Employee is
   subtype Name_T is String (1 .. 6);
   type Date_T is record
      Year  : Positive;
      Month : Positive;
      Day   : Positive;
   end record;
   type Job_T is (Sales, Engineer, Bookkeeping);

   ------------
   -- Person --
   ------------
   type Person_T is tagged record
      The_Name       : Name_T;
      The_Birth_Date : Date_T;
   end record;
   procedure Set_Name (O     : in out Person_T;
                       Value :        Name_T);
   function Name (O : Person_T) return Name_T;
   procedure Set_Birth_Date (O     : in out Person_T;
                             Value :        Date_T);
   function Birth_Date (O : Person_T) return Date_T;
   procedure Print (O : Person_T);

   --------------
   -- Employee --
   --------------
   type Employee_T is new Person_T with record
      The_Employee_Id : Positive;
      The_Start_Date  : Date_T;
   end record;
   not overriding procedure Set_Start_Date (O     : in out Employee_T;
                                            Value :        Date_T);
   not overriding function Start_Date (O : Employee_T) return Date_T;
   overriding procedure Print (O : Employee_T);

   --------------
   -- Position --
   --------------
   type Position_T is new Employee_T with record
      The_Job : Job_T;
   end record;
   not overriding procedure Set_Job (O     : in out Position_T;
                                     Value :        Job_T);
   not overriding function Job (O : Position_T) return Job_T;
   overriding procedure Print (O : Position_T);

end Employee;
```

Tagged Derivation Lab Solution - Types (Partial Body)

```
with Ada.Text_IO; use Ada.Text_IO;
package body Employee is

   function Image (Date : Date_T) return String is
     (Date.Year'Image & " -" & Date.Month'Image & " -" & Date.Day'Image);

   procedure Set_Name (O     : in out Person_T;
                       Value :        Name_T) is
   begin
      O.The_Name := Value;
   end Set_Name;
   function Name (O : Person_T) return Name_T is (O.The_Name);

   procedure Set_Birth_Date (O     : in out Person_T;
                             Value :        Date_T) is
   begin
      O.The_Birth_Date := Value;
   end Set_Birth_Date;
   function Birth_Date (O : Person_T) return Date_T is (O.The_Birth_Date);

   procedure Print (O : Person_T) is
   begin
      Put_Line ("Name: " & O.Name);
      Put_Line ("Birthdate: " & Image (O.Birth_Date));
   end Print;

   not overriding procedure Set_Start_Date
     (O     : in out Employee_T;
      Value :        Date_T) is
   begin
      O.The_Start_Date := Value;
   end Set_Start_Date;
   not overriding function Start_Date (O : Employee_T) return Date_T is
      (O.The_Start_Date);

   overriding procedure Print (O : Employee_T) is
   begin
      Put_Line ("Name: " & Name (O));
      Put_Line ("Birthdate: " & Image (O.Birth_Date));
      Put_Line ("Startdate: " & Image (O.Start_Date));
   end Print;

```

Tagged Derivation Lab Solution - Main

```
with Employee;
procedure Main is
   Applicant : Employee.Person_T;
   Employ    : Employee.Employee_T;
   Staff     : Employee.Position_T;

begin
   Applicant.Set_Name ("Wilma ");
   Applicant.Set_Birth_Date ((Year  => 1_234,
                              Month => 12,
                              Day   => 1));

   Employ.Set_Name ("Betty ");
   Employ.Set_Birth_Date ((Year  => 2_345,
                           Month => 11,
                           Day   => 2));
   Employ.Set_Start_Date ((Year  => 3_456,
                           Month => 10,
                           Day   => 3));

   Staff.Set_Name ("Bambam");
   Staff.Set_Birth_Date ((Year  => 4_567,
                          Month => 9,
                          Day   => 4));
   Staff.Set_Start_Date ((Year  => 5_678,
                          Month => 8,
                          Day   => 5));
   Staff.Set_Job (Employee.Engineer);

   Applicant.Print;
   Employ.Print;
   Staff.Print;
end Main;
```

Summary

Summary

Primitives rules for tagged types are trickier

- Primitives **forbidden** below freeze point
- **Unique** controlling parameter
- Tip: Keep the number of tagged types per package low

Supplement: Extensions With Privacy

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 => 1);
Mammal : Mammal_T := (Age            => 2,
                      Number_Of_Legs => 2);
Canine : Canine_T := (Age            => 2,
                      Number_Of_Legs => 4,
                      Domesticated   => True);
```

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

```
Mammal := (Animal with Number_Of_Legs => 4);
Canine := (Animal with Number_Of_Legs => 4,
                       Domesticated   => False);
Canine := (Mammal with Domesticated => True);
```

Private Tagged Types

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

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

- And still allow derivation

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

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

Private Extensions

- In the previous slide, we exposed the 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

```
Mammal := (Number_Of_Legs => 4,
           others         => <>);  -- Compile Error
```

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?

A. `function Create return Child_T is (Parents.Create with Count => 0);`
B. `function Create return Child_T is (others => <>);`
C. `function Create return Child_T is (0, 0);`
D. `function Create return Child_T is (P with Count => 0);`

Quiz

Given the following code:

```
package Parents is
  type Parent_T is tagged private;
  function Create return Parent_T;
private
  type Parent_T is tagged record
     Id : Integer;
  end record;
end Parents;

with Parents; use Parents;
package Children is
  P : Parent_T;
  type Child_T is new Parent_T with record
     Count : Natural;
  end record;
  function Create (C : Natural) return Child_T;
end Children;
```

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

A. *`function Create return Child_T is (Parents.Create with Count => 0);`*
B. `function Create return Child_T is (others => <>);`
C. `function Create return Child_T is (0, 0);`
D. *`function Create return Child_T is (P with Count => 0);`*

Explanations

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

Exceptions

Introduction

Rationale for Exceptions

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

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

Semantics Overview

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

Semantics Example: Raising

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

Semantics Example: Handling

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

Handler Part Is Skipped Automatically

- If no exceptions are active, returns normally

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

Handlers

Exception Handler Part

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

Example

```
begin
  Counter := Counter + 1;
exception
   when Constraint_Error =>
      Log_Error ("Overflow");
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

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

exception_handler ::=
  when [identifier:] exception_choice
                     {'|' exception_choice} =>
     sequence_of_statements

exception_choice ::= exception_name | others
```

Similarity to Case Statements

- Both structure and meaning
- Exception handler

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

- Case statement

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

Handlers Don't "Fall Through"

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

When an Exception Is Raised

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

- Caller

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

- Callee

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

Handling Specific Statements' Exceptions

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

Exception Handler Content

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

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

Quiz

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

What will get printed?

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

Quiz

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

What will get printed?

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

Explanations

A. `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. Line 6 does not raise an exception, (so `One` is not printed), but Line 2 does - causing `Two` to be printed. But Line 16 also raises an exception, causing `Three` to be printed
C. If we reach `Two`, the assignment on line 16 will cause `Three` to be reached
D. Divide by 0 on line 13 causes an exception, so `Two` must be called

Implicitly and Explicitly Raised Exceptions

Implicitly-Raised Exceptions

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

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

- Can happen by declaration elaboration

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

Some Language-Defined Exceptions

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

Explicitly-Raised Exceptions

Syntax

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

Raised by application via `raise` statements

- Named exception becomes active
- A `raise` by itself is only allowed in handlers

Example

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

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

User-Defined Exceptions

User-Defined Exceptions

Syntax

```
exception_declaration ::=
   identifier_list : exception
```

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

User-Defined Exceptions Example

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

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

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

Propagation

Propagation

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

Propagation Demo

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

```ada
begin -- Do_Something
  Maybe_Raise (3);
  Handled;
exception
  when Error =>
    Print ("Handle 3");
end Do_Something;
```

Termination Model

- When control goes to handler, it continues from here

```
procedure Joy_Ride is
begin
   loop
       Steer_Aimlessly;

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

Quiz

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

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

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

Quiz

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

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

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

Explanations

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

Exceptions As Objects

Exceptions Are Not Objects

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

But You Can Treat Them As Objects

- For raising and handling, and more
- Standard Library

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

Exception Occurrence

- Syntax associates an object with active exception

```
exception_handler ::=
  when [identifier:] exception_choice
                     {'|' exception_choice} =>
     sequence_of_statements

exception_choice ::= exception_name | others
```

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

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

Exception_Occurrence Query Functions

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

Exception ID

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

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

Raise Expressions

Raise Expressions

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

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

Lab

Exceptions Lab

Numeric String Verifier

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

Exceptions Lab Solution - Numeric Types

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

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

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

```
package body Numeric_Types is

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

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

end Numeric_Types;
```

Exceptions Lab Solution - Main

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

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

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

Summary

Exceptions Are Not Always Appropriate

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

Relying on Exception Raising Is Risky

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

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

- Recommended

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

Summary

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

Interfacing with C

Introduction

Introduction

- Lots of C code out there already
  - Maybe even a lot of reusable code in your own repositories
- Need a way to interface Ada code with existing C libraries
  - Built-in mechanism to define ability to import objects from C or export Ada objects
- Passing data between languages can cause issues
  - Sizing requirements
  - Passing mechanisms (by reference, by copy)

Import / Export

Import / Export Aspects (1/2)

- Aspects `Import` and `Export` allow Ada and C to interact
  - `Import` indicates a subprogram imported into Ada
  - `Export` indicates a subprogram exported from Ada
- Need aspects definining calling convention and external name
  - `Convention => C` tells linker to use C-style calling convention
  - `External_Name => "<name>"` defines object name for linker
- Ada implementation

```
procedure Imported_From_C with
   Import,
   Convention    => C,
   External_Name => "SomeProcedureInC";

procedure Exported_To_C with
   Export,
   Convention    => C,
   External_Name => "some_ada_procedure";
```

- C implementation

```
void SomeProcedureInC (void) {
   // some code
}

extern void ada_some_procedure (void);
```

Import / Export Aspects (2/2)

- You can also import/export variables
  - Variables imported won't be initialized
  - Ada view

```
My_Var : Integer_Type with
   Import,
   Convention    => C,
   External_Name => "my_var";
```

  - C implementation

```
int my_var;
```

Import / Export with Pragmas

- You can also use **pragma** to import/export entities

```
procedure C_Some_Procedure;
pragma Import (C, C_Some_Procedure, "SomeProcedure");

procedure Some_Procedure;
pragma Export (C, Some_Procedure, "ada_some_procedure");
```

Parameter Passing

Parameter Passing to/from C

- The mechanism used to pass formal subprogram parameters and function results depends on:
    - The type of the parameter
    - The mode of the parameter
    - The Convention applied on the Ada side of the subprogram declaration
- The exact meaning of *Convention C*, for example, is documented in *LRM* B.1 - B.3, and in the *GNAT User's Guide* section 3.11.

Passing Scalar Data As Parameters

- C types are defined by the Standard
- Ada types are implementation-defined
- GNAT standard types are compatible with C types
    - Implementation choice, use carefully
- At the interface level, scalar types must be either constrained with representation clauses, or coming from Interfaces.C
- Ada view

```
with Interfaces.C;
function C_Proc (I : Interfaces.C.Int)
    return Interfaces.C.Int;
pragma Import (C, C_Proc, "c_proc");
```

- C view

```
int c_proc (int i) {
  /* some code */
}
```

Passing Structures As Parameters

- An Ada record that is mapping on a C struct must:
  - Be marked as convention C to enforce a C-like memory layout
  - Contain only C-compatible types
- C View

```
enum Enum {E1, E2, E3};
struct Rec {
   int A, B;
   Enum C;
};
```

- Ada View

```
type Enum is (E1, E2, E3) with Convention => C;
type Rec is record
  A, B : int;
  C : Enum;
end record with Convention => C;
```

- This can also be done with pragmas

```
type Enum is (E1, E2, E3);
Pragma Convention (C, Enum);
type Rec is record
  A, B : int;
  C : Enum;
end record;
Pragma Convention (C, Rec);
```

Parameter Modes

- `in` scalar parameters passed by copy
- `out` and `in out` scalars passed using temporary pointer on C side
- By default, composite types passed by reference on all modes except when the type is marked `C_Pass_By_Copy`
    - Be very careful with records - some C ABI pass small structures by copy!
- Ada View

```
Type R1 is record
   V : int;
end record
with Convention => C;

type R2 is record
   V : int;
end record
with Convention => C_Pass_By_Copy;
```

- C View

```
struct R1{
   int V;
};
struct R2 {
   int V;
};
void f1 (R1 p);
void f2 (R2 p);
```

Complex Data Types

Unions

- C `union`

```
union Rec {
   int A;
   float B;
};
```

- C unions can be bound using the `Unchecked_Union` aspect
- These types must have a mutable discriminant for convention purpose, which doesn't exist at run-time
  - All checks based on its value are removed - safety loss
  - It cannot be manually accessed
- Ada implementation of a C `union`

```
type Rec (Flag : Boolean := False) is
record
   case Flag is
      when True =>
         A : int;
      when False =>
         B : float;
   end case;
end record
with Unchecked_Union,
     Convention => C;
```

Arrays Interfacing

- In Ada, arrays are of two kinds:
  - Constrained arrays
  - Unconstrained arrays
- Unconstrained arrays are associated with
  - Components
  - Bounds
- In C, an array is just a memory location pointing (hopefully) to a structured memory location
  - C does not have the notion of unconstrained arrays
- Bounds must be managed manually
  - By convention (null at the end of string)
  - By storing them on the side

Arrays From Ada to C

- An Ada array is a composite data structure containing 2 parts: Bounds and Components
  - **Fat pointers**
- When arrays can be sent from Ada to C, C will only receive an access to the components of the array
- Ada View

```
type Arr is array (Integer range <>) of int;
procedure P (V : Arr; Size : int);
pragma Import (C, P, "p");
```

- C View

```
void p (int * v, int size)  {
}
```

Arrays From C to Ada

- There are no boundaries to C types, the only Ada arrays that can be bound must have static bounds
- Additional information will probably need to be passed
- Ada View

```
-- DO NOT DECLARE OBJECTS OF THIS TYPE
type Arr is array (0 .. Integer'Last) of int;

procedure P (V : Arr; Size : int);
pragma Export (C, P, "p");

procedure P (V : Arr; Size : int) is
begin
   for J in 0 .. Size - 1 loop
      -- code;
   end loop;
end P;
```

- C View

```
extern void p (int * v, int size);
int x [100];
p (x, 100);
```

Strings

- Importing a `String` from C is like importing an array - has to be done through a constrained array
- `Interfaces.C.Strings` gives a standard way of doing that
- Unfortunately, C strings have to end by a null character
- Exporting an Ada string to C needs a copy!

```
Ada_Str : String := "Hello World";
C_Str : chars_ptr := New_String (Ada_Str);
```

- Alternatively, a knowledgeable Ada programmer can manually create Ada strings with correct ending and manage them directly

```
Ada_Str : String := "Hello World" & ASCII.NUL;
```

- Back to the unsafe world - it really has to be worth it speed-wise!

Interfaces.C

Interfaces.C Hierarchy

- Ada supplies a subsystem to deal with Ada/C interactions
- `Interfaces.C` - contains typical C types and constants, plus some simple Ada string to/from C character array conversion routines
  - `Interfaces.C.Extensions` - some additional C/C++ types
  - `Interfaces.C.Pointers` - generic package to simulate C pointers (pointer as an unconstrained array, pointer arithmetic, etc)
  - `Interfaces.C.Strings` - types / functions to deal with C `"char *"`

Interfaces.C

```
package Interfaces.C is

   --  Declaration's based on C's <limits.h>
   CHAR_BIT  : constant := 8;
   SCHAR_MIN : constant := -128;
   SCHAR_MAX : constant := 127;
   UCHAR_MAX : constant := 255;

   type int   is new Integer;
   type short is new Short_Integer;
   type long  is range -(2 ** (System.Parameters.long_bits - Integer'(1)))
     .. +(2 ** (System.Parameters.long_bits - Integer'(1))) - 1;

   type signed_char is range SCHAR_MIN .. SCHAR_MAX;
   for signed_char'Size use CHAR_BIT;

   type unsigned       is mod 2 ** int'Size;
   type unsigned_short is mod 2 ** short'Size;
   type unsigned_long  is mod 2 ** long'Size;

   type unsigned_char is mod (UCHAR_MAX + 1);
   for unsigned_char'Size use CHAR_BIT;

   type ptrdiff_t is range -(2 ** (System.Parameters.ptr_bits - Integer'(1))) ..
                           +(2 ** (System.Parameters.ptr_bits - Integer'(1)) - 1);

   type size_t is mod 2 ** System.Parameters.ptr_bits;

   --  Floating-Point
   type C_float     is new Float;
   type double      is new Standard.Long_Float;
   type long_double is new Standard.Long_Long_Float;

   type char is new Character;
   nul : constant char := char'First;

   function To_C   (Item : Character) return char;
   function To_Ada (Item : char)      return Character;

   type char_array is array (size_t range <>) of aliased char;
   for char_array'Component_Size use CHAR_BIT;

   function Is_Nul_Terminated (Item : char_array) return Boolean;

   -- (more not specified here)

end Interfaces.C;
```

Interfaces.C.Extensions

```
package Interfaces.C.Extensions is

   --  Definitions for C "void" and "void *" types
   subtype void     is System.Address;
   subtype void_ptr is System.Address;

   --  Definitions for C incomplete/unknown structs
   subtype opaque_structure_def is System.Address;
   type opaque_structure_def_ptr is access opaque_structure_def;

   --  Definitions for C++ incomplete/unknown classes
   subtype incomplete_class_def is System.Address;
   type incomplete_class_def_ptr is access incomplete_class_def;

   --  C bool
   type bool is new Boolean;
   pragma Convention (C, bool);

   --  64-bit integer types
   subtype long_long is Long_Long_Integer;
   type unsigned_long_long is mod 2 ** 64;

   -- (more not specified here)

end Interfaces.C.Extensions;
```

Interfaces.C.Pointers

```
generic
   type Index is (<>);
   type Component is private;
   type Component_Array is array (Index range <>) of aliased Component;
   Default_Terminator : Component;

package Interfaces.C.Pointers is

   type Pointer is access all Component;
   for Pointer'Size use System.Parameters.ptr_bits;

   function Value (Ref        : Pointer;
                   Terminator : Component := Default_Terminator)
                   return Component_Array;
   function Value (Ref    : Pointer;
                   Length : ptrdiff_t)
                   return Component_Array;

   Pointer_Error : exception;

   function "+" (Left : Pointer;   Right : ptrdiff_t) return Pointer;
   function "+" (Left : ptrdiff_t; Right : Pointer)   return Pointer;
   function "-" (Left : Pointer;   Right : ptrdiff_t) return Pointer;
   function "-" (Left : Pointer;   Right : Pointer)   return ptrdiff_t;

   procedure Increment (Ref : in out Pointer);
   procedure Decrement (Ref : in out Pointer);

   -- (more not specified here)

end Interfaces.C.Pointers;
```

Interfaces.C.Strings

```
package Interfaces.C.Strings is

   type char_array_access is access all char_array;
   for char_array_access'Size use System.Parameters.ptr_bits;

   type chars_ptr is private;

   type chars_ptr_array is array (size_t range <>) of aliased chars_ptr;

   Null_Ptr : constant chars_ptr;

   function To_Chars_Ptr (Item      : char_array_access;
                          Nul_Check : Boolean := False) return chars_ptr;

   function New_Char_Array (Chars : char_array) return chars_ptr;

   function New_String (Str : String) return chars_ptr;

   procedure Free (Item : in out chars_ptr);

   function Value (Item : chars_ptr) return char_array;
   function Value (Item   : chars_ptr;
                   Length : size_t)
                   return char_array;
   function Value (Item : chars_ptr) return String;
   function Value (Item   : chars_ptr;
                   Length : size_t)
                   return String;

   function Strlen (Item : chars_ptr) return size_t;

   -- (more not specified here)

end Interfaces.C.Strings;
```

Lab

Interfacing with C Lab

- Requirements
  - Given a C function that calculates speed in MPH from some information, your application should
    - Provide some values for distance and time (consider hard-coding, or prompting for user input)
    - Populate the structure appropriately
    - Call C function to return speed
    - Print speed to console
- Hints
  - Structure contains the following components
    - Distance (floating point)
    - Distance Type (enumeral)
    - Seconds (floating point)

Interfacing with C Lab - GNAT Studio

To compile/link the C file into the Ada executable:

1. Make sure the C file is in the same directory as the Ada source files
2. Edit → Project Properties
3. Sources → Languages → Check the "C" box
4. Build and execute as normal

Interfacing with C Lab Solution - Ada

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

   package Float_Io is new Ada.Text_IO.Float_IO (Interfaces.C.C_float);

   One_Minute_In_Seconds : constant := 60.0;
   One_Hour_In_Seconds   : constant := 60.0 * One_Minute_In_Seconds;

   type Distance_T is (Feet, Meters, Miles) with Convention => C;
   type Data_T is record
      Distance      : Interfaces.C.C_float;
      Distance_Type : Distance_T;
      Seconds       : Interfaces.C.C_float;
   end record with Convention => C_Pass_By_Copy;
   function C_Miles_Per_Hour (Data : Data_T) return Interfaces.C.C_float
      with Import, Convention => C, External_Name => "miles_per_hour";

   Object_Feet : constant Data_T :=
     (Distance => 6_000.0,
      Distance_Type => Feet,
      Seconds  => Interfaces.C.C_float(One_Minute_In_Seconds));
   Object_Meters : constant Data_T :=
     (Distance => 3_000.0,
      Distance_Type => Meters,
      Seconds  => Interfaces.C.C_float(One_Hour_In_Seconds));
   Object_Miles : constant Data_T :=
     (Distance => 1.0,
      Distance_Type =>
      Miles, Seconds => 1.0);

   procedure Run (Object : Data_T) is
   begin
      Float_Io.Put (Object.Distance);
      Put (" " & Distance_T'Image (Object.Distance_Type) & " in ");
      Float_Io.Put (Object.Seconds);
      Put (" seconds = ");
      Float_Io.Put (C_Miles_Per_Hour (Object));
      Put_Line (" mph");
   end Run;

begin
   Run (Object_Feet);
   Run (Object_Meters);
   Run (Object_Miles);
end Main;
```

Interfacing with C Lab Solution - C

```
enum DistanceT { FEET, METERS, MILES };
struct DataT {
    float distance;
    enum DistanceT distanceType;
    float seconds;
    };

float miles_per_hour (struct DataT data) {
   float miles = data.distance;
   switch (data.distanceType) {
      case METERS:
         miles = data.distance / 1609.344;
         break;
      case FEET:
         miles = data.distance / 5280.0;
         break;
   };
   return miles / (data.seconds / (60.0 * 60.0));
}
```

Summary

Summary

- Possible to interface with other languages (typically C)
- Ada provides some built-in support to make interfacing simpler
- Crossing languages can be made safer
  - But it still increases complexity of design / implementation

Tasking

Introduction

Concurrency - One Program, Many Things Happening

- **Sequential programs** - one instruction at a time
- **Concurrent programs** - multiple activities *conceptually* happening at once
  - Even on one CPU
  - Think many cooks in one kitchen
- Why concurrency?
  - Respond to external events (real-time systems)
  - Improve performance or responsiveness

Concurrency in Ada

- Built-in language constructs
  - Not a library - part of the semantics
- `task` - concurrent process
  - Compiler/runtime coordinate all tasks (not programmer)
- `protected` - safe shared data access

Process Communication

- Tasks can
  - Rendezvous with other tasks
    - Via `entry` call
    - Data can be passed like in subprograms
  - Wait for another task
  - Block other tasks
- Protected objects control data access
  - Multiple simultaneous readers
  - One writer at a time
    - All other accesses blocked during write

Tasks

Basic Task

- **Specification** (`task`)
  - What other parts of the program see
- **Body** (`task body`)
  - Code the task actually runs

```
procedure Main is
  task My_Task; -- declare the task

  task body My_Task is -- implement the task
  begin
     Put_Line ("Entered My_Task");
  end My_Task;
begin
  Put_Line ("In Main");
end Main;
```

- `My_Task` starts automatically when `Main` starts

> **Note**
> The application's main program is itself a task

Basic Synchronization

- Enclosing scope cannot exit until all of its tasks have completed

```
procedure Show_Simple_Sync is
  task Hello_Task;
  task body Hello_Task is
  begin
    for Counter in 1 .. 10 loop
      Put_Line ("hello");
    end loop;
  end Hello_Task;
begin
  null;
  --  Will wait here until Hello_Task is finished
end Show_Simple_Sync;
```

Rendezvous

- Tasks synchronize actions using mechanism called *rendezvous*
  - Follows a client/server model
  - **Server** task declares an `entry`
    - Public point of synchronization other tasks call
  - **Client** task calls that entry same as a procedure
  - **Server** must then accept call
- Both tasks are blocked until rendezvous is complete
  - **Server** must perform entry processing
  - **Client** is waiting for **Server** to finish

```
task Server_Task is
  entry Receive_Message (S : in String);
end Server_Task;

task body Server_Task is
begin
  accept Receive_Message (S : in String) do -- waiting for client
    Put_Line ("Received: " & S);
  end Receive_Message; -- release to client
end Server_Task;

procedure Client is
begin
  -- The client calls the entry and waits
  Server_Task.Receive_Message ("Hello!");
end Client;
```

Sequential Rendezvous

- Task can have multiple entry points that need to be called in sequence
- Each entry call is blocking

```
task body Worker is
  Job_Data : Some_Data_Type;
  Result   : Some_Result_Type;
begin
  loop
    -- Step 1: Wait for a client to provide a new job
    accept Get_Work (Data : in Some_Data_Type) do
      Job_Data := Data;
    end Get_Work;

    -- Step 2: Do the work (details omitted)
    Result := Process (Job_Data);

    -- Step 3: Wait for the client to request the result
    accept Report_Result (Final_Result : out Some_Result_Type) do
      Final_Result := Result;
    end Report_Result;
  end loop;
end Worker;
```

```
Worker.Get_Work (My_Job);           -- Give the worker a job
Worker.Report_Result (My_Result);   -- Get the result
```

- Worker cannot generate report until after `Get_Work` has completed

Selective Rendezvous

- Task isn't limited to waiting for just one entry
  - Typically, **server** task needs to be able to accept several kinds of requests
- To wait for multiple entries at the same time use `select` statement
  - Task waits until **client** calls an `entry` included in `select`, then executes that block
  - If multiple calls waiting, the runtime chooses which **client** to handle
    - Selection order is not guaranteed

Select Example in Code

- **Server** task waits for either a message to process or a signal to stop

```
task body Controller is
begin
  loop
    -- Wait for EITHER Receive_Message OR Stop to be called
    select
      accept Receive_Message (V : in String) do
        Put_Line ("Processing: " & V);
      end Receive_Message;
    or
      accept Stop;
        Put_Line ("Stopping task...");
        exit; -- Exit the loop to terminate the task
    end select;
  end loop;
end Controller;
```

- How a **client** would use it:

```
-- Client_X
Controller.Receive_Message ("Run diagnostic");

-- Client_Y
Controller.Stop;
```

  - `Client_X` and `Client_Y` can be the same task, different tasks, or the main program

Quiz

```
task Simple_Task is
   entry Go;
end Simple_Task;

task body Simple_Task is
begin
    accept Go do
        loop
            null;
        end loop;
    end Go;
end Simple_Task;
```

What happens when `Simple_Task.Go` is called?

A. Compilation error
B. Run-time error
C. The calling task completes successfully
D. Simple_Task hangs

Quiz

```
task Simple_Task is
   entry Go;
end Simple_Task;

task body Simple_Task is
begin
    accept Go do
        loop
            null;
        end loop;
    end Go;
end Simple_Task;
```

What happens when `Simple_Task.Go` is called?

A. Compilation error
B. Run-time error
C. The calling task completes successfully
D. ***Simple_Task hangs***

A. Syntax is correct
B. Code is doing what it is supposed to
C. Caller must wait for `Go` block to finish
D. `Go` block is entered, but never completes

Protected Objects

The Problem: Sharing Data is Dangerous!

- What happens if two tasks try to update the same variable at the exact same time?
  - Task 1 reads the value X (it's 10)
  - Task 2 reads the value X (it's also 10)
  - Task 1 calculates 10 + 5 and writes 15 back to X
  - Task 2 calculates 10 + 1 and writes 11 back to X
- The first update is lost!
  - This is a *race condition*
- Race condition
  - Leads to corrupt and unpredictable data
  - Protected objects prevent concurrent modifications

The Solution: Protected Objects

- Protected object is designed for safe, concurrent access to shared data
  - Acts as a monitor, guarding the data it holds
  - Has a restricted set of operations
    - Can't manipulate its data directly
  - Guarantees concurrency-safe semantics
    - Prevents concurrent modifications from corrupting data

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 Objects - Declaration

```
package Protected_Objects is

   protected Object is

      procedure Set (Prompt : String; V : Integer);
      function Get (Prompt : String) return Integer;

   private
      Local : Integer := 0;
   end Object;

end Protected_Objects;
```

Example: Protected Objects - Body

```
with Ada.Text_IO; use Ada.Text_IO;

package body Protected_Objects is

   protected body Object is

      procedure Set (Prompt : String; V : Integer) is
         Str : constant String := "Set " & Prompt & V'Image;
      begin
         Local := V;
         Put_Line (Str);
      end Set;

      function Get (Prompt : String) return Integer is
         Str : constant String := "Get " & Prompt & Local'Image;
      begin
         Put_Line (Str);
         return Local;
      end Get;

   end Object;

end Protected_Objects;
```

Quiz

```
protected Counter is
   procedure Initialize (V : Integer);
   procedure Increment;
   function Decrement return Integer;
   function Query return Integer;
private
   Object : Integer := 0;
end Counter;
```

Which completion(s) of `Counter` is (are) illegal?

A.
```
procedure Initialize (V : Integer) is
begin
  Object := V;
end Initialize;
```

B.
```
procedure Increment is
begin
  Object := Object + 1;
end Increment;
```

C.
```
function Decrement return Integer is
begin
  Object := Object - 1;
  return Object;
end Decrement;
```

D.
```
function Query return Integer is begin
  return Object;
end Query;
```

Quiz

```ada
protected Counter is
   procedure Initialize (V : Integer);
   procedure Increment;
   function Decrement return Integer;
   function Query return Integer;
private
   Object : Integer := 0;
end Counter;
```

Which completion(s) of `Counter` is (are) illegal?

A.
```ada
procedure Initialize (V : Integer) is
begin
  Object := V;
end Initialize;
```

B.
```ada
procedure Increment is
begin
  Object := Object + 1;
end Increment;
```

C.
```ada
function Decrement return Integer is
begin
  Object := Object - 1;
  return Object;
end Decrement;
```

D.
```ada
function Query return Integer is begin
  return Object;
end Query;
```

A. Legal - Assignment to protected data allowed in procedure
B. Legal - subprograms do not need parameters
C. Functions in a protected object cannot modify protected data
D. Legal - Reading of protected data allowed in function

Delays

Delay Keyword

- `delay` keyword part of tasking
- Blocks for a time
    - Measured in seconds
    - Resolution dependent on runtime
        - Typically can assume at least 0.1 seconds resolution
- **Relative:** Blocks for at least `Duration`
- **Absolute:** Blocks until no earlier than `Calendar.Time` or `Real_Time.Time`

Delay Example

```
with Ada.Calendar; use Ada.Calendar;
with Ada.Text_IO;  use Ada.Text_IO;
procedure Main is
   Start_Time : Time := Clock;
   function Time_Str return String is
     (Duration'Image (Clock-Start_Time)(1 .. 5));
   task Relative;
   task body Relative is
   begin
      for Counter in 1 .. 5 loop
         delay 0.1;
         Put_Line (Time_Str &
                    " => Relative " & I'Image);
      end loop;
   end Relative;
begin
   for Counter in 1 .. 5 loop
      delay until Start_Time + Duration (I) * 0.1;
      Put_Line (Time_Str &
                 " => Absolute " & I'Image);
   end loop;
end Main;
```

Output

```
0.10 => Relative 1
0.10 => Absolute 1
0.21 => Absolute 2
0.21 => Relative 2
0.30 => Absolute 3
0.33 => Relative 3
0.41 => Absolute 4
0.45 => Relative 4
0.51 => Absolute 5
0.56 => Relative 5
```

Task and Protected Types

Beyond One-Off Tasks: Task and Protected Types

- Creating templates for tasks and protected objects
  - When you need multiple tasks or protected objects that behave similarly
- Task (and protected) types rather than objects
  - Can be parameterized to cause different behavior

Reusable Task Patterns

```
-- Simple task that, upon startup, loops forever
-- calling some procedure and pausing
task type Worker is
   entry Initialize (Cycle : Duration);
end Worker;

task body Worker is
   Delay_Time : Duration;
begin
   -- Wait until initialized with a delay time
   accept Initialize (Cycle : Duration) do
       Delay_Time := Cycle;
   end Initialize;
   -- Once task has started, just wait a certain
   -- amount of time and then call a procedure
   loop
      delay Delay_Time;
      Do_Something;
   end loop;
end Worker;

--  Two tasks that start at elaboration and wait for initialization
Worker_1, Worker_2 : Worker;

procedure Main is
begin
   -- Start the tasks at different frequencies
   Worker_1.Initialize (1.0);
   Worker_2.Initialize (2.0);
end Main;
```

- Each Worker runs its own independent thread of control

Reusable Protected Components

- *Protected type*
  - Defines synchronized access to shared data
- *Protected object*
  - An instance of a protected type
- Procedures and functions inside the type control access rules

```
protected type Counter is
   procedure Increment;
   function Value return Integer;
private
   Count : Integer := 0;
end Counter;

protected body Counter is
   procedure Increment is
   begin
      Count := Count + 1;
   end Increment;

   function Value return Integer is (Count);
end Counter;

C1, C2 : Counter;
```

Some Advanced Concepts

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 Some_Task_T is ...
type Some_Task_Ptr_T is access Some_Task_T;

task body Some_Task_T is ...
...
declare
   Task_Object    : Some_Task_T;    -- Task_Object starts
   Access_To_Task : Some_Task_Ptr_T;
begin
   Access_To_Task := new Some_Task_T;
   -- Task pointed to by Access_To_Task starts
```

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 Task_Definition is
   task type One_Second_Timer;
end Task_Definition;

package body Task_Definition is
   task body One_Second_Timer is
      loop
         delay 1.0;
         Put_Line ("tick");
      end loop;
   end One_Second_Timer;

   Task_Instance : One_Second_Timer;
end Task_Definition;
```

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 its identifying information to a monitor periodically based on the delay length
  - Create a protected object that stores the identifying information of the task that called it
  - Main program should periodically check the protected object, and print when it detects a task switch
    - I.e. If the current task is different than the last printed task, print the identifying information for the current task

Tasking Lab Solution - Protected Object

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

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

Tasking Lab Solution - Task Type

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

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

Tasking Lab Solution - Main

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

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

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

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

end Main;
```

Summary

Summary

- Tasks are **language-based** concurrency mechanisms
  - Typically implemented as threads
  - Not necessarily for **truly** parallel operations
  - Originally for task-switching / time-slicing
- Multiple mechanisms to **synchronize** tasks
  - Delay
  - Rendezvous
  - Protected Objects
- Protected objects also control access to data
  - Only one writer at a time

Subprogram Contracts

Introduction

Design-By-Contract

- Source code acting in roles of **client** and **supplier** under a binding **contract**
  - *Contract* specifies *requirements* or *guarantees*
    - *"A specification of a software element that affects its use by potential clients."* (Bertrand Meyer)
  - *Supplier* provides services
    - Guarantees specific functional behavior
    - Has requirements for guarantees to hold
  - *Client* utilizes services
    - Guarantees supplier's conditions are met
    - Requires result to follow the subprogram's guarantees

Ada Contracts

- Ada contracts include enforcement
  - At compile-time: specific constructs, features, and rules
  - At run-time: language-defined and user-defined exceptions
- Facilities as part of the language definition
  - Range specifications
  - Parameter modes
  - Generic contracts
  - 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**
  - Failure to meet a contract is an *assertion failure*
    - Typically causes an `Assertion_Error` exception to be raised

Preconditions and Postconditions

Subprogram-based Assertions

- **Explicit** part of a subprogram's **specification**
  - Unlike defensive code
- *Precondition*
  - Assertion expected to hold **prior to** subprogram call
- *Postcondition*
  - Assertion expected to hold **after** subprogram return
- Requirements and guarantees on both supplier and client
- Syntax uses **aspects**

```
procedure Push (This : in out Stack_T;
                Value : Content_T)
  with Pre  => not Full (This),
       Post => not Empty (This)
               and Top (This) = Value;
```

Requirements and Guarantees

- We want to call procedure `Turn_On` from this code

```
procedure Start is
begin
   Initialize;
   Turn_On;
   Proceed;
```

- `Turn_On` has the following specification

```
procedure Turn_On;
  with Pre  => Has_Power,
       Post => System_Ready;
```

- Which means
  - `Turn_On` *requires* power, so `Start` must *guarantee* it
  - `Start` *requires* the system to be ready, so `Turn_On` *guarantees* it

Defensive Programming

- Should be replaced by subprogram contracts when possible

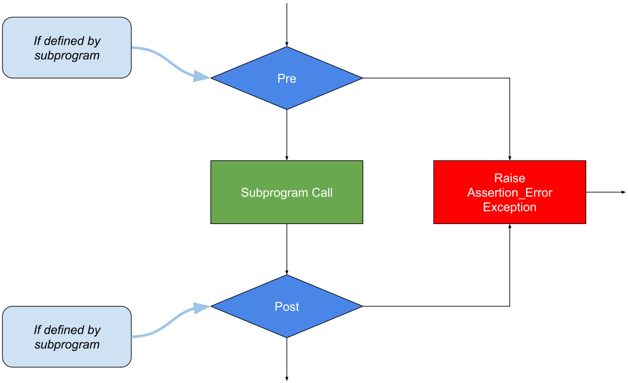
```
procedure Push (The_Stack : Stack) is
   Entry_Length : constant Positive := Length (The_Stack);
begin
   pragma Assert (not Is_Full (The_Stack)); -- entry condition
   [...]
   pragma Assert (Length (The_Stack) = Entry_Length + 1); -- exit condition
end Push;
```

- 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 Example

- Multiple aspects separated by commas

```
procedure Push (This : in out Stack;
                Value : Content)
  with Pre  => not Full (This),
       Post => not Empty (This) and Top (This) = Value;
```

(Sub)Types Allow Simpler Contracts

- Pre-condition

```
procedure Compute_Square_Root (Input : Integer;
                               Result : out Natural)
  with Pre  => Input >= 0,
       Post => (Result * Result) <= Input and
               (Result + 1) * (Result + 1) > Input;
```

- Subtype

```
procedure Compute_Square_Root (Input  : Natural;
                               Result : out Natural)
   with
       -- "Pre => Input >= 0" not needed
       -- (Input can't be < 0)
       Post => (Result * Result) <= Input and
               (Result + 1) * (Result + 1) > Input;
```

Preventing Exceptions with ... Exceptions?

```
function Area (Length : Positive;
               Height : Positive)
               return Positive is
   (Length * Height);
```

- We want to prevent an exception when we calculate some area
  - So we should make sure the multiplication doesn't overflow, right?

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

- But what happens when we verify the precondition?
  - We do the math anyways, causing an exception!
- Better solution

```
function Area (Length : Positive;
               Height : Positive)
               return Positive is
with Pre => Positive'Last / Height <= Length;
```

Quiz

```
-- 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?

A. "0" is printed
B. Constraint Error exception
C. Assertion Error exception
D. Undefined behavior

Quiz

```
-- 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?

A. "0" is printed
B. Constraint Error exception
C. ***Assertion Error exception***
D. Undefined behavior

Explanations

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

Special Attributes

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 'Old

```
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 before call
   Global (Index'Old)
      -- Global and Index after call
      = Global (Index);
```

Error on Conditional Evaluation of 'Old

- This code is **incorrect**

```
procedure Clear_Character (In_String : in out String;
                           At_Position : Positive)
   with Post => (if At_Position in In_String'Range
                 then In_String (At_Position)'Old = ' ');
```

- 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

```
procedure Clear_Character (In_String : in out String;
                           At_Position : Positive)
   with Post => (if At_Position in In_String'Range
                 then In_String'Old (At_Position) = ' ');
```

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
procedure Set_And_Move (Value :        Character;
                        Index : in out Index_T);
   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)`
- `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
procedure Set_And_Move (Value :        Character;
                        Index : in out Index_T);
   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`

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
procedure Set_And_Move (Value :        Character;
                        Index : in out Index_T);
   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`

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
procedure Set_And_Move (Value :        Character;
                        Index : in out Index_T);
   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 (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;
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

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

```ada
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)`

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)));
```

Postconditions and Expression Functions

- Expression functions use their implementation as a postcondition **only if**
  - Spec and body are in the same scope (or there is no spec)
  - There is no postcondition specified

```
function Add (L, R : Integer) return Integer is
   (L + R);
-- Postcondition is "Add'Result = L + R"

function Subtract (L, R : Integer) return Integer;
-- No postcondition specified (so postcondition is "True")

function Multiply (L, R : Integer) return Integer
  with Post => Multiply'Result in Integer'Range;
-- Postcondition just says the function didn't fail
```

Contracts Code Reuse

- Contracts are about **usage** and **behaviour**
  - Not optimization
  - Not implementation details
  - **Abstraction** level is typically high
- Extracting them to `function` is a good idea
  - *Code as documentation, executable specification*
  - Completes the **interface** that the client has access to
  - Allows for **code reuse**

```
procedure Withdraw (This   : in out Account;
                    Amount :        Currency) with
  Pre  => Open (This) and then Funds_Available (This, Amount),
  Post => Balance (This) = Balance (This)'Old - Amount;
...
function Funds_Available (This   : Account;
                          Amount : Currency)
                          return Boolean is
   (Amount > 0.0 and then Balance (This) >= Amount)
  with Pre => Open (This);
```

- A `function` may be unavoidable
  - Referencing private type components

Subprogram Contracts on Private Types

```
package Bank is
  type Account is private;
  procedure Process_Transaction (This : Account) with
    Pre => This.Balance > 0; -- no visibility to Balance
  ...
  function Current_Balance (This : Account) return Integer;
  ...
  procedure R (This : Account) with
    Pre => Current_Balance (This) > 0; -- using accessor is fine
  ...
private
  type Account is record
    Balance : Natural;
    ...
  end record;
  function Current_Balance (This : Account) return Integer is
      (This.Balance);
end Bank;
```

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

```
type Stack (Capacity : Positive) is tagged private;
procedure Push (This : in out Stack;
                Value : Content) with
  Pre  => not Full (This);
```

- 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

```
procedure Push (This : in out Stack;
                Value : Content) with
  Pre  => not Full (This) or else raise Overflow_Error;
```

- *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_rm/implementation_defined_pragmas.html#pragma-assertion-policy

- Certain advantage over explicit checks which are **harder** to disable
  - Conditional compilation via global `constant Boolean`

```
procedure Push (This : in out Stack;  Value : Content) is
begin
  if Debugging then
    if Full (This) then
      raise Overflow;
    end if;
  end if;
```

Lab

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 → Project Properties
    - **Build** → **Switches** → **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;
private
   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 =>
           Queue.Entries (Index).Priority >=
           Queue.Entries (Index - 1).Priority));
end Priority_Queue;
```

Subprogram Contracts Lab Solution - Queue (Body)

```ada
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);
      else
         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
      Value      := Queue.Entries (Queue.Size).Value;
      Queue.Size := Queue.Size - 1;
   end Pop;

end Priority_Queue;
```

Subprograms Contracts Lab Solution - Main

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Priority_Queue;
procedure Main is
   Queue : Priority_Queue.Queue_T;
   Value : Priority_Queue.String_T;
begin

   Ada.Text_IO.Put_Line ("Normal processing");
   for Count in 1 .. 3 loop
      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
      Queue.Pop (Value);
      Put_Line (Value);
   end loop;

   Ada.Text_IO.Put_Line ("Test overflow");
   for Count in 1 .. 4 loop
      for Priority in Priority_Queue.Priority_T'Range loop
         Queue.Push (Priority, Priority'Image & Count'Image);
      end loop;
   end loop;

end Main;
```

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
- Note that pre/postconditions can be used on concurrency constructs too

	Clients	Suppliers
Preconditions	Requirement	Guarantee
Postconditions	Guarantee	Requirement

- Contracts are just another form of *defensive coding*
  - But can't replace all of it
  - And it's still code, so it could be wrong!

Type Contracts

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

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

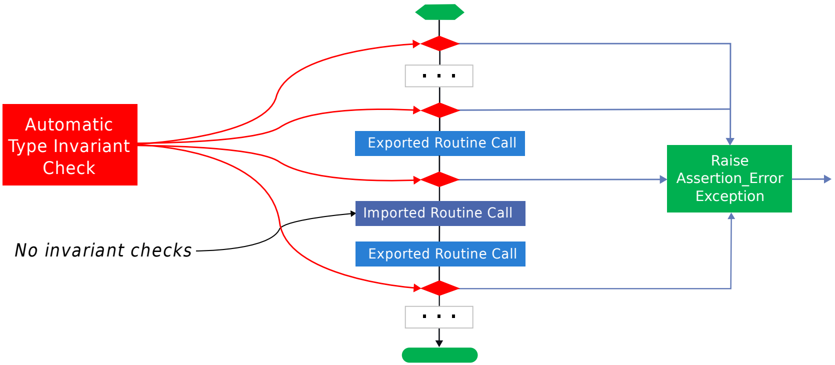
```
subtype Weekdays is Days range Mon .. Fri;

-- 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

Invariant Over Object Lifetime (Calls)



Example Type Invariant

- A bank account balance must always be consistent
  - Consistent Balance: Total Deposits - Total Withdrawals = Balance

```
package Bank is
  type Account is private with
    Type_Invariant => Consistent_Balance (Account);
  ...
  -- Called automatically for all Account objects
  function Consistent_Balance (This : Account)
    return Boolean;
  ...
private
  ...
end Bank;
```

Invariants Don't Apply Internally

- No checking within supplier package
  - Otherwise there would be no way to implement anything!
- Only matters when clients can observe state

```
procedure Open (This : in out Account;
                Name : in String;
                Initial_Deposit : in Currency) is
begin
  This.Owner := To_Unbounded_String (Name);
  This.Current_Balance := Initial_Deposit;
  -- invariant would be false here!
  This.Withdrawals := Transactions.Empty_Vector;
  This.Deposits := Transactions.Empty_Vector;
  This.Deposits.Append (Initial_Deposit);
  -- invariant is now true
end Open;
```

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
      return (This = 0);
   end Zero;
end Operations;
```

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!

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

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 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

```
type_declaration ::=
     type identifier [discriminant_part] is type_definition
        [aspect_specification];

subtype_declaration ::=
   subtype identifier is subtype_indication
        [aspect_specification];

aspect_specification ::=
   with aspect_mark => expression

aspect_mark ::= Static_Predicate | Dynamic_Predicate
```

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 ("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

```
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;
```

- 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;
```

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

Dynamic Predicate Expression Content

- Any arbitrary Boolean expression
  - Hence all allowed static predicates' content
- Plus additional operators, etc.

```
subtype Even is Integer
  with Dynamic_Predicate => Even mod 2 = 0;
subtype Vowel is Character with Dynamic_Predicate =>
  (case Vowel is
   when 'A' | 'E' | 'I' | 'O' | 'U' => True,
   when others => False);
```

Note

`when others` is evaluated at run-time, so this predicate must be **dynamic**

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

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;
```

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_day := sat;
      when sat =>
        exit;
      when others =>
        a_day := weekend'succ (a_day);
    end case;
  end loop;
end;
```

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)

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;
```

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)));
```

- 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
  - Then you get a compile error

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. Static predicate is allowed when condition is a static membership test
B. `if` statement not allowed in a predicate
C. Function call not allowed in `Static_Predicate` (this would be OK for `Dynamic_Predicate`)
D. Missing parentheses around `case` expression

Lab

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
  - To enable assertions in the runtime from GNAT STUDIO
    - Edit → Project Properties
    - **Build** → **Switches** → **Ada**
    - Click on *Enable assertions*

Type Contracts Lab Solution - Schedule (Spec)

```ada
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;
                        Day        :        Days_T;
                        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
   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 => ' ');
      Day        : Days_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);
end Schedule;
```

Type Contracts Lab Solution - Schedule (Body)

```
with Ada.Text_IO; use Ada.Text_IO;
package body Schedule is

   procedure Add_Class
     (Classes    : in out Classes_T;
      Name       :        Name_T;
      Day        :        Days_T;
      Start_Time :        Time_T;
      End_Time   :        Time_T) is
   begin
      Classes.Size                := Classes.Size + 1;
      Classes.List (Classes.Size) :=
        (Name       => Name, Day => Day,
         Start_Time => Start_Time, End_Time => End_Time);
   end Add_Class;

   procedure Print (Classes : Classes_T) is
   begin
      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) & " )");
      end loop;
   end Print;

end Schedule;
```

Type Contracts Lab Solution - Main

```ada
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  ",
                      Day        => Mon,
                      Start_Time => 10.0,
                      End_Time   => 11.0);
   Classes.Add_Class (Name       => "History   ",
                      Day        => Tue,
                      Start_Time => 11.0,
                      End_Time   => 12.5);
   Classes.Add_Class (Name       => "Biology   ",
                      Day        => Wed,
                      Start_Time => 13.0,
                      End_Time   => 14.0);
   Classes.Print;
   begin
      Classes.Add_Class (Name       => "Chemistry ",
                         Day        => Thu,
                         Start_Time => 13.0,
                         End_Time   => 14.0);
   exception
      when The_Err : others =>
         Put_Line (Exception_Information (The_Err));
   end;
end Main;
```

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

Summary

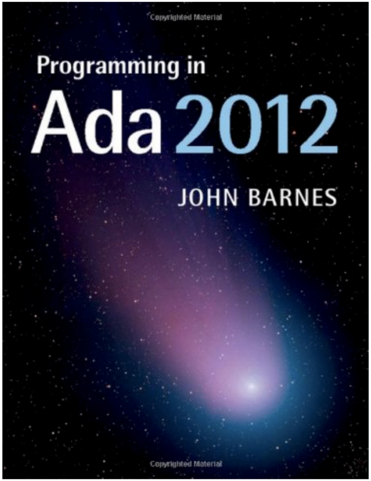
- Type Invariants are valid at external boundary
  - Useful for complex types - type may not be consistent during an operation
- Predicates are like other constraint checks
  - Checked on declaration, assignment, calls, etc

Annex - Reference Materials

General Ada Information

Learning the Ada Language

- Written as a tutorial for those new to Ada



Reference Manual

- **LRM** - Language Reference Manual (or just **RM**)
  - Always on-line (including all previous versions) at www.adaic.org
- Finding stuff in the RM
  - You will often see the RM cited like this **RM 4.5.3(10)**
  - This means *Section 4.5.3, paragraph 10*
  - Have a look at the table of contents
    - Knowing that chapter 5 is *Statements* is useful
  - Index is very long, but very good!

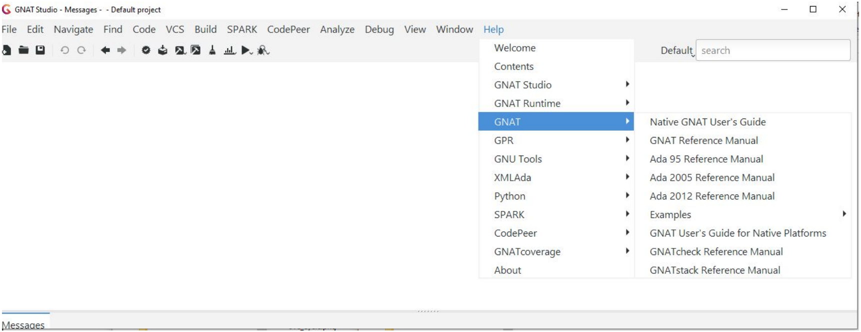
Current Ada Standard

- "ISO/IEC 8652(E) with Technical Corrigendum 1"
- Useful as a Reference Text but not intended to be read from beginning to end

GNAT-Specific Help

Reference Manual

- Reference Manual(s) available from GNAT STUDIO Help



GNAT Tools

- GNAT User's Guide
  - LOTS of info about the main tools: the GNAT compiler, binder, linker etc.
- GNAT Reference Manual
  - How GNAT implements Ada, pragmas, aspects, attributes etc. etc.
- GNAT STUDIO (the IDE)
  - Tutorial
  - User's Guide
  - Release notes
- Many other tools

AdaCore Support

Need More Help?

- If you have an AdaCore subscription:
  - Find out your customer number #XXXX
- Open a "Case" via the GNATtracker web interface and/or email
  - GNATtracker
    - Select "Create A New Case" from the main landing page
  - Email
    - Send to: support@adacore.com
    - Subject should read: #XXXX - (descriptive text)
- Not just for "bug reports"
  - Ask questions, make suggestions, etc.

GNAT in Practice

File Naming Conventions

Default File Naming Conventions

- GNAT compiler assumes one compilable entity per file
  - Package specification, subprogram body, etc
- File names should match the name of the compilable entity
  - Replace `.` with `-`
- File extensions describe the usage
  - **.ads** → Specification / Interface
  - **.adb** → Body / Implementation

Example Filenames with Contents

```
package Some_Types is
   type One_T is new Integer;
   type Two_T is new Character;
   function Convert (Src : One_T)
                     return Two_T;
end Some_Types;
```

Package specification for Some_Types *is in file* `some_types.ads`

```
package body Some_Types is
   function Convert (Src : One_T)
                     return Two_T is
      (Two_T'Val (Integer (Src)));
end Some_Types;
```

Package body for Some_Types *is in file* `some_types.adb`

```
function Some_Types.Child (Src : Two_T)
                           return One_T;
```

Subprogram specification for function Child *which is a child of* Some_Types *is in file* `some_types-child.ads`

Converting to GNAT Naming Conventions

- Use GNATCHOP to convert file containing Ada code to GNAT names
  - If file contains multiple units, will generate multiple files
  - `-w` is the most common switch - will overwrite existing files
  - Can specify destinaton directory
    - If not specified, files created in same directory are source
- Files for standard library units created using `-k` switch
  - **krunch** - generated filename has no more than 8 characters
  - `Ada.Characters` → `a-charac.ads`
  - Historical reasons (i.e "8.3" filenames)

Using Other Naming Conventions

- Sometimes you don't want to change filenames
  - Sharing source across multiple compilers
  - Different versions of a file based on build parameters
- Controlled via package `Naming` in project file
  - **Example:** your source files use `.1.ada` for specs and `.2.ada` for bodies

```
package Naming is
   for Spec_Suffix ("Ada") use ".1.ada";
   for Body_Suffix ("Ada") use ".2.ada";
end Naming;
```

  - Example: different implementations for different platforms

```
package Naming is
   case Platform is
      when "windows =>
         for Body ("My_IO") use "my_io.windows";
      when "linux" =>
         for Body ("My_IO") use "my_io.linux";
end Naming;
```

More Information

For further information, see Section 3.3 *File Naming Topics and Utilities* in the **GNAT User's Guide**

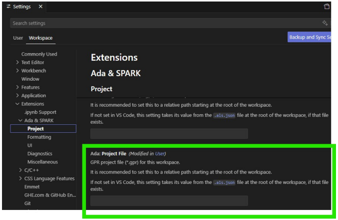
Using VS Code with GNAT

Setting up VS Code

- Need to download and install *Ada & SPARK extensions for VS Code*
  - Visual Studio Marketplace
    (https://marketplace.visualstudio.com/items?itemName=AdaCore.ada)
  - Search for **adacore** in VS Code Extensions (Ctrl+Shift+X)
- Make sure GNAT is installed and on your path
  - If not already downloaded, look on GitHub
    (https://github.com/AdaCore/ada_language_server/tree/master/integration/vscode/ada#getting-additional-tools)

Using VS Code with the Labs

- From a command prompt
  - Navigate to the appropriate folder (`prompt` or `answer`)
  - Enter `code .`
    - Explorer tree should show `default.gpr` and Ada file(s)
- Set GPR file via **Settings**
  - File → Preferences → Settings
  - In **Settings** window, switch to Workspace tab, click Extensions/Ada & SPARK/Project and enter `default.gpr`



Building the Lab

- Use VS Code predefined task `ada: Build current project`
  - Go to the **Command Palette** (Ctrl+Shift+B)
  - Search for ada commands - you're looking for ada: Build current project
  - Press Enter to run the task
    - Select View → Problems to see build output