Subprogram Contracts

Introduction

Contract-Based Programming

- Source code acting in roles of **client** and **supplier** under a binding **contract**
  - *Supplier* provides services
  - *Client* utilitizes services
  - *Contract* specifies requirements and guarantees
    - "A specification of a software element that affects its use by potential clients." (Bertrand Meyer)
- Includes enforcement
  - At compile-time: specific constructs, features, and rules
  - At run-time: language-defined and user-defined exceptions

Contracts In Ada

- Exist implicitly in facilities you may already be using
  - Exceptions
  - Range specifications
  - Subtypes
  - Parameter modes
  - OOP interface types
  - et cetera
- Are explicitly supported
  - Low-level and high-level **assertions**
  - Predicates
  - Including OOP context

Terminology

Assertion	Boolean expression expected to be True (Said "to hold" when True)
Precondition	Assertion expected to hold prior to client call
Postcondition	Assertion expected to hold after supplier return
Predicate	Assertion expected to hold for all objects of given type
Invariant	Assertion expected to hold for all objects of given ADT when viewed by clients

Low-Level Assertions

- Language-defined package with procedures
  - Raise `Assertion_Error` if expression is False

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

- Language-defined pragma
  - Easier to enable/disable
  - Definition

```
pragma Assert (any_boolean_expression [, [Message =>] string_expression]);
```

  - Usage

```
procedure Push ( Value : in     Content_T ) is
begin
  pragma Assert (not Full (Stack));
  -- if we get here, stack is not full
  ...
```

High-Level Assertions

- Pre- and postconditions specify obligations on supplier and client

```
procedure Push (This : in out Stack_T;
                Value : Content_T)
  with Pre  => not Full (This),          -- requirement
       Post => not Empty (This)          -- guarantee
               and Top (This) = Value;
```

- Type invariants ensure properties of objects over their lifetimes
  - *Described in a different module*

```
type Table_T is private with Type_Invariant =>
  Sorted (Table_T); -- user-defined boolean expression
-- external usage of Table will always be sorted
function Sorted (This : Table_T) return Boolean;
```

Preconditions and Postconditions

Examples

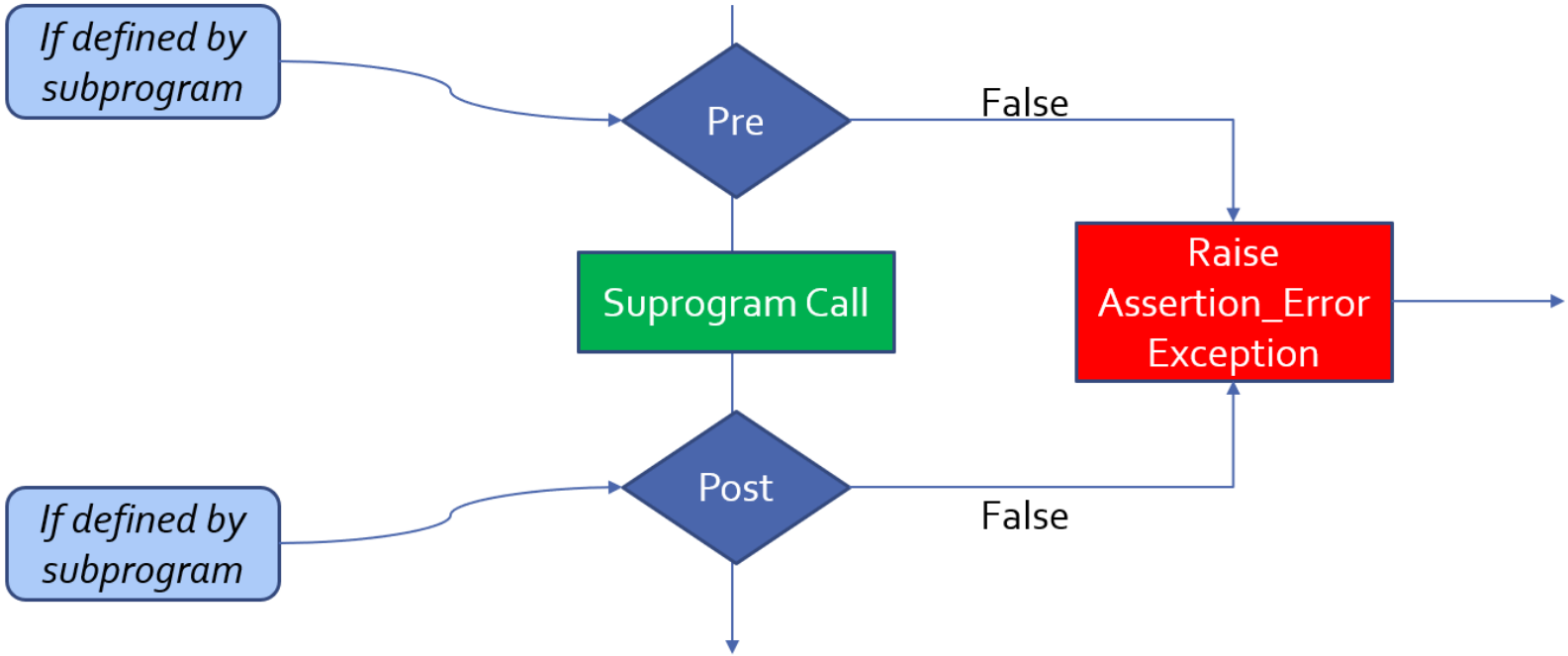
```
package Stack_Pkg is
   procedure Push (Item : in Integer) with
         Pre  => not Full,
         Post => not Empty and then Top = Item;
   procedure Pop (Item : out Integer) with
         Pre  => not Empty,
         Post => not Full;
   function Pop return Integer with
         Pre  => not Empty,
         Post => not Full;
   function Top return Integer with
         Pre => not Empty;
   function Empty return Boolean;
   function Full return Boolean;
end Stack_Pkg;

package body Stack_Pkg is
   Values  : array (1 .. 100) of Integer;
   Current : Natural := 0;

   -- Push/Pop cannot fail because preconditions prevent it
   procedure Push (Item : in Integer) is
   begin
      Current          := Current + 1;
      Values (Current) := Item;
   end Push;

   procedure Pop (Item : out Integer) is
   begin
      Item    := Values (Current);
      Current := Current - 1;
   end Pop;

   function Pop return Integer is
      Item : constant Integer := Values (Current);
   begin
      Current := Current - 1;
      return Item;
   end Pop;

   function Top return Integer is (Values (Current));
   function Empty return Boolean is (Current not in Values'Range);
   function Full return Boolean is (Current >= Values'Length);
end Stack_Pkg;
```

Pre/Postcondition Contracts

- Suppliers provide subprograms, clients call them
- Supplier will:
  - Guarantee specific functional behavior
  - Specify conditions required for guarantees to hold
- Client will:
  - Ensure supplier's conditions are met
  - Rely on resulting guarantees
- Obligations and guarantees are enforced
  - At run-time
  - Under user control

Pre/Postcondition Semantics

- Calls inserted automatically by compiler



Pre/Postcondition Placement

- Contracts referenced by subprogram bodies
  - Requirements to provide service
  - Guarantee on results
- But used by clients so appear with declarations
  - Typically separate declarations in package specs
  - On subprogram body when no separate declaration used
- Spec and body

```
procedure Op with Pre => ... ;
procedure Op is
   ...
```

- Body only

```
procedure Op with Pre => ...
is
   ...
```

Expressions In Pre/Postconditions

- Add to expressive power
- Contract value is a Boolean
- Can include any legal Ada expression

```
type List is array (1 .. 10) of Integer;
procedure Extract_and_Clear (From : in out List;
                             K : integer;
                             Value : out Integer)
  with Post => (if K in List'Range then From(K) = 0);
```

Contract with Quantified Expression

```
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;
```

Preconditions

- Define obligations on client for successful call
  - Precondition specifies required conditions
  - Clients must meet precondition for supplier to succeed
- Boolean expressions
  - Arbitrary complexity
  - Specified via aspect name **Pre**
- Checked prior to call by client
  - `Assertion_Error` raised if false

```
procedure Push (This : in out Stack;  Value : Content)
  with Pre => not Full (This);
```

Precondition Content

- Any parameter of the subprogram
  - Any mode
- Any visible name in scope
  - Variables, including globals
  - Functions, often expression functions
  - Can refer to functions not yet defined
    - 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;
```

Postconditions

- Define obligations on supplier
  - Specify guaranteed conditions after call
- Boolean expressions (same as preconditions)
  - Specified via aspect name **Post**
- Content as for preconditions, plus some extras
- Checked after corresponding subprogram call
  - `Assertion_Error` raised if false

```
procedure Push (This : in out Stack;  Value : Content)
  with Pre  => not Full (This),
       Post => not Empty (This) and Top (This) = Value;
...
function Top (This : Stack) return Content
  with Pre => not Empty (This);
```

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;
```

Quiz

```
function Area (L : Integer; H : Integer) return Integer is
   (L * H)
with Pre => ?
```

Which expression will guarantee `Area` calculates the correct result for all values `L` and `H`

A. `Pre => L > 0 and H > 0`
B. `Pre => L < Integer'last and H < Integer'last`
C. `Pre => L * H in Integer`
D. None of the above

Quiz

```
function Area (L : Integer; H : Integer) return Integer is
   (L * H)
with Pre => ?
```

Which expression will guarantee `Area` calculates the correct result for all values `L` and `H`

A. `Pre => L > 0 and H > 0`
B. `Pre => L < Integer'last and H < Integer'last`
C. `Pre => L * H in Integer`
D. ***None of the above***

Explanations

A. Does not handle large numbers
B. Does not handle negative numbers
C. Will generate a constraint error on large numbers

The correct precondition would be

```
L > 0 and then H > 0 and then Integer'Last / L <= H
```

to prevent overflow errors on the range check.

Special Attributes

Examples

```
package Stack_Pkg is
   procedure Push (Item : in Integer) with
         Pre  => not Full,
         Post => not Empty and then Top = Item;
   procedure Pop (Item : out Integer) with
         Pre  => not Empty,
         Post => not Full 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;

   -- Push/Pop cannot fail because preconditions prevent it
   procedure Push (Item : in Integer) is
   begin
      Current          := Current + 1;
      Values (Current) := Item;
   end Push;

   procedure Pop (Item : out Integer) is
   begin
      Item    := Values (Current);
      Current := Current - 1;
   end Pop;

   function Pop return Integer is
      Item : constant Integer := Values (Current);
   begin
      Current := Current - 1;
      return Item;
   end Pop;

   function Top return Integer is (Values (Current));
   function Empty return Boolean is (Current not in Values'Range);
   function Full return Boolean is (Current >= Values'Length);
end Stack_Pkg;
```

Referencing Previous Values In Postconditions

- Values as they were just before the call
- Uses language-defined attribute **'Old**
  - Can be applied to most any visible object
    - Makes a copy so `limited` types not supported
  - Applied to formal parameters, typically

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

- Copies can be expensive!

Example for Attribute 'Old

- Simple code to shift a character in a string

```
function At_Index (Index : Integer) return Character is
   (Global (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 =>
   -- call At_Index before call
   At_Index (Index)'Old
      -- look at Index position in Global before call
      = Global'Old (Index'Old)
   and
   -- call At_Index after call with original Index
   At_Index (Index'Old)
      -- look at Index position in Global after call
      = Global (Index);
```

What Happens When 'Old Is Evaluated

- Copy made on entrance for use by postconditions
- "Safety" checks in postcondition weren't applied to the entrance copy evaluation
- Incorrect

```
procedure Clear_Character (In_String : in out String;
                           Look_For  : in     Character;
                           Found_At  :    out Integer)
   with Post => Found_At in In_String'Range and
                In_String (Found_At'Old) = Look_For;
```

  - On entry, **Found_At** is not valid, so `In_String(Found_At'Old)` will likely raise an exception

- Solution (required)

```
procedure Clear_Character (In_String : in out String;
                           Look_For  : in     Character;
                           Found_At  :    out Integer)
   with Post => Found_At in In_String'Range and
        In_String'Old(Found_At) = Look_For;
```

Using Function Results In Postconditions

- Sometimes you need to reference to the value returned by function you are defining
- Uses language-defined attribute **'Result**

```
function Greatest_Common_Denominator (A, B : Integer)
  return Integer with
    Pre  =>  A > 0 and B > 0,
    -- pass result of Greatest_Common_Denominator to Is_GCD
    Post =>  Is_GCD (A,
                     B,
                     Greatest_Common_Denominator'Result);

function Is_GCD (A, B, Candidate : Integer)
    return Boolean is (... );
```

- Only applicable to functions, in postconditions

Quiz

```
type Index_T is range 1 .. 100;
-- Database initialized such that value for element at I = I
Database : array (Index_T) of Integer;
-- Set the value for element Index to Value and
-- then increment Index by 1
function Set_And_Move (Value :        Integer;
                       Index : in out Index_T)
                       return Boolean
   with Post => ...
```

What would the following expressions evaluate to in the Postcondition when called with `Value` of -1 and `Index` of 10?

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

Quiz

```
type Index_T is range 1 .. 100;
-- Database initialized such that value for element at I = I
Database : array (Index_T) of Integer;
-- Set the value for element Index to Value and
-- then increment Index by 1
function Set_And_Move (Value :        Integer;
                       Index : in out Index_T)
                       return Boolean
   with Post => ...
```

What would the following expressions evaluate to in the Postcondition when called with `Value` of -1 and `Index` of 10?

Database'Old(Index)	11	Use new index in copy of original Database
Database(Index'Old)	-1	Use copy of original index in current Database
Database(Index)'Old	10	Evaluation of Database(Index) before call

In Practice

Pre/Postconditions: To Be or Not To Be

- Preconditions generally not too expensive
  - Reasonable default for checking
    - But they can be disabled at run-time!
- Postconditions can be comparatively expensive
  - Use of **'Old** and **'Result** involve copying (maybe deep)
- Enabling preconditions alone makes sense when calling trusted library routines
  - That way, you catch client errors
- Do you enable them all the time? It depends...
  - How tight is the overall timing in your application?
  - Is response-time available to respond to violations?
  - What are the consequences of not catching violations?
  - How expensive are run-time checks in this implementation?

No Secret Precondition Requirements

- Should only require what client can ensure
  - By only referencing entities also available to clients
- Language rules enforce this

```
package P is
   type Bar is private;
   ...
   function Foo (This : Bar) return Baz
     with Pre => Hidden; -- illegal reference
private
   function Hidden return Boolean;
   ...
end P;
```

Postconditions Are Good Documentation

```
procedure Reset
    (Unit : in out DMA_Controller;
     Stream : DMA_Stream_Selector)
  with Post =>
    not Enabled (Unit, Stream) and
    Operating_Mode (Unit, Stream) = Normal_Mode and
    Selected_Channel (Unit, Stream) = Channel_0 and
    not Double_Buffered (Unit, Stream) and
    Priority (Unit, Stream) = Priority_Low and
    (for all Interrupt in DMA_Interrupt =>
        not Interrupt_Enabled (Unit, Stream, Interrupt));
```

Postcondition Limitations

- Sometimes cannot specify all relevant properties without repeating body
  - Unlike preconditions

```
function Greatest_Common_Denominator (A, B : Integer)
  return Integer with
  Pre  =>  A > 0 and B > 0,
  Post =>  Is_GCD (A, B, Greatest_Common_Denominator'Result);
function Is_GCD (A, B, Candidate : Integer)
    return Boolean is
  (A rem Candidate = 0 and
   B rem Candidate = 0 and
   (for all K in 1 .. Integer'Min (A,B) =>
      (if (A rem K = 0 and B rem K = 0)
       then K <= Candidate)));
```

Use Functions In Pre/Postconditions

- Abstraction increases chances of getting it right
  - Provides higher-level interface to clients too

```
procedure Withdraw (This   : in out Account;
                    Amount :        Currency) with
  Pre  => Open (This) and Funds_Available (This, Amount),
  Post => Balance (This) = Balance (This)'Old - Amount;
...
function Funds_Available (This   : Account;
                          Amount : Currency)
                          return Boolean is
    (Amount > 0.0 and then Balance (This) >= Amount)
  with Pre => Open (This);
```

- May be unavoidable
  - Cannot reference hidden components of private types in the package visible part

Private Part Reference Approach

```
package P is
  type T is private;
  procedure Q (This : T) with
    Pre => This.Total > 0; -- not legal
  ...
  function Current_Total (This : T) return Integer;
  ...
  procedure R (This : T) with
    Pre => Current_Total (This) > 0; -- legal
  ...
private
  type T is record
    Total : Natural ;
    ...
  end record;
  function Current_Total (This : T) return Integer is
      (This.Total);
end P;
```

Using Pre/Postconditions

- Assertions are not good logic control structures
  - Use `if` or `case` in subprogram to handle special cases
- Assertions are not good external input validation
  - Contracts are internal: between parts of the source code
  - Precondition cannot prevent invalid user data entry
- Precondition violations indicate client bugs
  - Maybe the requirements spec is wrong, but too late to argue now
- Postcondition violations indicate supplier bugs

Preconditions Or Explicit Checks?

- Logically part of the spec so should be textually too
  - Otherwise clients must examine the body, breaking abstraction
- Do this

```
type Stack (Capacity : Positive) is tagged private;
procedure Push (This : in out Stack;
                Value : Content) with
  Pre  => not Full (This);
```

- Or do this

```
procedure Push (This : in out Stack;
                Value : Content) is
begin
  if Full (This) then
    raise Overflow;
  end if;
  ...
```

- But not both
  - A subprogram body should never test its own preconditions

Advantages Over Explicit Checks

- Pre/postconditions can be turned off
  - Like language-defined checks
- Explicit checks cannot be disabled except by changing the source text
  - Conditional compilation via preprocessor (`#ifdef`)
  - Conditional compilation via static Boolean constants

```
procedure Push (This : in out Stack;  Value : Content) is
begin
  if Debugging then
    if Full (This) then
      raise Overflow;
    end if;
  end if;
  ...
end Push;
```

Exceptions

Controlling the Exception Raised

- Failing pre/postconditions raise `Assertion_Error`
- Abstractions may define dedicated exceptions
  - Assertion Error

```
type Stack (Capacity : Positive) is tagged private;
procedure Push (This : in out Stack;  Value : Content) with
  Pre  => not Full (This);
```

  - Overflow

```
procedure Push (This : in out Stack;  Value : Content) is
begin
  if Full (This) then
    raise Overflow;
  end if;
  ...
```

- How to get them raised in preconditions?
  - Not needed for postconditions (failures are supplier bugs)

"Raise Expressions" In Preconditions

```
package Bounded_Stacks is
  type Stack (Capacity : Positive) is tagged private;
  Overflow, Underflow : exception;
  procedure Push (This  : in out Stack;
                  Value : in     Content) with
    Pre  => not Full (This)
            or else raise Overflow; -- raise this exception
  procedure Pop (This  : in out Stack;
                 Value :    out Content) with
    Pre  => not Empty (This)
            or else raise Underflow; -- raise this exception
  function Empty (This : Stack) return Boolean;
  function Full (This : Stack) return Boolean;
...
private
...
end Bounded_Stacks;
```

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 run-time from GNAT Studio
    - Edit → Project Properties
    - **Build** → **Switches** → **Ada**
    - Click on *Enable assertions*

Subprogram Contracts Lab Solution - Queue (Spec)

```
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
package Priority_Queue is
   Overflow : exception;
   type Priority_T is (Low, Medium, High);
   type Queue_T is tagged private;

   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 Unbounded_String) 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    : Unbounded_String;
   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)

```
package body Priority_Queue is

   procedure Push (Queue    : in out Queue_T;
                   Priority :        Priority_T;
                   Value    :        String) is
      Last      : Size_T renames Queue.Size;
      New_Entry : Entries_T := (Priority, To_Unbounded_String (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 Unbounded_String) is
   begin
      Value      := Queue.Entries (Queue.Size).Value;
      Queue.Size := Queue.Size - 1;
   end Pop;

end Priority_Queue;
```

Subprograms Contracts Lab Solution - Main

```
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
with Ada.Text_IO;           use Ada.Text_IO;
with Priority_Queue;
procedure Main is
   Queue : Priority_Queue.Queue_T;
   Value : Unbounded_String;
begin

   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 (To_String (Value));
   end loop;

   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 obligations/benefits
- Enhances readability and understandability
  - Specification contains explicitly expressed properties of code
- Improves testability but also likelihood of passing!
- Aids in debugging
- Facilitates tool-based analysis
  - Compiler checks conformance to obligations
  - Static analyzers (e.g., SPARK, CodePeer) can verify explicit precondition and postconditions

Summary

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

	Clients	Suppliers
Preconditions	Obligation	Guarantee
Postconditions	Guarantee	Obligation