Australian National University

STRUCTURED PROGRAMMING

Week 1-2 Laboratory for Concurrent and Distributed Systems

Uwe R. Zimmer based on material by Alistair Rendell

Pre-Laboratory Checklist

- ❑ **You have a solid background in sequential computer systems and their programming. If not: reconsider your course choice now!**
- ❑ **You have read the complete lab text before you enter the lab.**
- ❑ **You can design small to medium sized software systems based on imperative as well as functional programming languages.**
- ❑ **You have a basic understanding of computer processors.**
- ❑ **You have a firm grasp on static type systems and can use them to your advantage.**

Objectives

The objective of this first lab is to gain some familiarity with Ada as your x'th language. The emphasis will be on the differences which Ada offers in comparison to languages which you might be more used to at the moment – like Haskell, C, Python, Java, Assembler, Bash, JavaScript, C++, PHP, C#, MatLab, or any of the other languages which you indicated in the poll.

Are there other languages with strong support for concurrency? Of course, and in fact we will use some of those later. The landscape of programming languages is currently is an even more dynamic phase than usual, and recent languages like e.g. Rust, Chapel, or the enhancements in C++14 are all very well worth your time to investigate further. Older concurrent languages like e.g. Erlang are also still valid and very educational.

Ada (besides being a concurrent language) emphasises on dependability and maintainability while keeping all controls at your disposal. So it is neither Haskell nor C, yet it offers the nagging compiler messages of Haskell (in fact a lot more, and much more readable ones) as well as the direct control and speed of C (in fact sometimes even more). Ada runs many compile-time checks and can (if you wish so) add also many run-time checks to keep systems dependable and well monitored at all times. The high-integrity and real-time systems background of Ada (which makes it one of the dominant languages in e.g. the aviation industry) shines through in many places while you will be working with this language.

The key issue is though that any language can only check against what you provide. So it is essential that your programs are highly specific and express everything you know about your systems. This first lab will guide you through some of the basic syntax and semantic to get there.

The lab will end in an early outlook into what happens if you take things to a concurrent level.

Exercise 1: Hello Concurrent World

This is a familiarization exercise to warm you up and make you design your first program in this lab from an empty sheet. The job is simply: calculate the arithmetic and harmonic mean of a bunch of numbers:

$$\textit{Arithmetic Mean}\,(A) := n^{-1}\sum_{a\in A} a$$
$$\textit{Harmonic Mean}\,(A) := n\Big(\sum_{a\in A} a^{-1}\Big)^{-1}$$

with $A = \{a_1, \ldots, a_n\}$

Download the empty project framework `Hello_Concurrent_World` from the course site. I will provide the code snippets in this course as zip files which contain all project definitions to open them directly with gps (the Gnat Programming Studio). This saves you lots of time and gives you the comfort of a full featured Integrated Development Environment (IDE) without the pain of needing to set up projects or compiler options. For the die-hard emacs, vi and Notepad fans among you: please look into the "Sources" directory of those projects and open them with anything your heart desires, followed by a command line compilation – do not forget to set at least the most basic warnings in this case though. The rest of us navigate into the directory where the project file is and type "`gps&`" to power up a copy of gps which automatically loads the current project.

This is a shockingly empty project with one source file called `means.adb` which happens to be empty as well. This is where you start coding.

If you are comfortable already, start programming, calculate those two means of some set of constants and print your results on the terminal. You can skip reading the rest of this exercise in this case.

Ah, ok, you are still here? Let me give you some pointers then.

If you look at the examples in the lectures, you will find that the top level executable (or "main" if your coming from Planet C) is a `procedure` in Ada. So you will need to define such a procedure. The compiler expects that the name of the main procedure matches the file name it lives in, so that would make it `Means`,

Defining a bunch of numbers is straight forward:

```
Test_Floats : constant array (1 .. 3) of Float := (1.0, 2.0, 3.0);
```

but let's be professional about things from the start and rather define some types instead:

```
type Number  is digits 5;
type Numbers is array (Positive range <>) of Number;

Test_Numbers : constant Numbers := (10.0, 20.0, 30.0, 40.0);
```

You life became much easier already, as you can now define a function from `Numbers` to `Number`, which works on any size array. We also don't care to define indices explicitly or work explicitly on indices as Ada provides you of course with the means to work on elements in an array without knowing their array indices, and even if you later on really need to know the actual indices for something, you can always refer to the attributes of the array at hand. You are now also explicit about the precision of your numbers and leave it to the compiler to figure out whether this can be done or how to implement this efficiently (try to request a numeric precision which is beyond what your current CPU can reasonably process and see what the compiler tells you).

For the actual calculations you will eventually find yourself in a situation where you need to divide numbers. Easy enough, but Ada will insist that the left and the right hand side of this operation has matching types. If you for instance take the lengths of your `Numbers` set in form of `Test_Numbers'Length`, you will gain a discrete number which won't divide with a real number. So you will need to allow Ada explicitly to convert into a real number by saying for example `Number (Test_Numbers'Length)`.

Putting anything on the screen will require that you make some basic IO operations visible first. This is achieved by importing:

```
with Ada.Text_IO; use Ada.Text_IO;
```

at the very beginning of your program file. You can then already put things on the screen using the `'Image` attribute which came automatically with your definition of `Number`, e.g. as:

```
Put ("Arithmetic_Mean : "); Put (Number'Image (A_Mean (Test_Numbers))); New_Line;
```

You have no control on how your numbers look like that way though and so there is a more complex option too. If you declare your own `IO` package for your `Number` type (you can only declare this after you declared your `Number` type itself – Ada insists on declaring things in order):

```
package Number_Float_IO is new Float_IO (Number);
use Number_Float_IO;
```

then you can use more fine grained `IO` options like:

```
Put (A_Mean (Test_Numbers), 2, Number'Digits - 3, 0);
```

or if you like it verbose, you can also use the names of those parameters (instead of relying on their position) and say:

```
Put (Item => A_Mean (Test_Numbers),
     Fore => 2,
     Aft  => Number'Digits - 3,
     Exp  => 0);
```

Now we expressed for example that we don't trust the last digit of our results considering that the precision of the original numbers was only 5 digits to begin with.

At this point you have enough hints to implement the rest of your means calculating program. Obviously for a real world version of this, you would consider to implement this as a module which can be later on used by other programs. Ada will give you any option you like to implement this, like function pointers or generic packages or dynamic dispatching etc.. So keep in mind that this is just your hello world program and you will work more real world structures in the next exercise.

Fun extension: you are calculating two different means – like in: one after the other. You signed up for a course on concurrency, right? Soo … what do you need to make this happen concurrently?

Just declare two tasks:

```
task Arithmetic_Mean;
task Harmonic_Mean;
```

and then provide some code which is supposed to be run inside those tasks (distribute the code which you have at the moment over two tasks):

```
task body Arithmetic_Mean is
begin
   …
end Arithmetic_Mean;
task body Harmonic_Mean is
begin
   …
end Harmonic_Mean;
```

Done. When you execute your program next time, you will actually use two of your CPUs (in fact three, but that's a topic which we will come back to soon).

Submit your new zip file for tests and code review by us on the *SubmissionApp* under "Lab 1 Means".

Interlude: Split specifications and implementations

This is a trivial (yet essential) aspect as you are experienced in structured programming. What form of structured programming is your preferred flavour (objects, classes, modules, templates, interfaces, etc.) does not make any difference at the moment, as it is all about encapsulation and information hiding at this point in the course. For your own benefit it is essential though that you know and can utilize the differences between those different approaches.

Every module (called a `package` in Ada) in your system will have a specification part and an implementation part. Informally, the former will define what you can access and how you can use it, while the latter will express how it will work. Obviously a user of a module only needs to know how to use something (and hopefully what the involved computational complexities are), but not how it is actually coded. In Ada those two aspects are separated into two separate source files which will be independently compiled.

Let's have a look at a simple specification for a queue in Ada:

```
package Queue_Pack_Private is

   Queue_Size : constant Positive := 10;

   type Element is range 1_000 .. 24_000;
   type Queue_Type is limited private;

   procedure Enqueue (Item :     Element; Queue : in out Queue_Type);
   procedure Dequeue (Item : out Element; Queue : in out Queue_Type);

   function Is_Empty (Queue : Queue_Type) return Boolean;
   function Is_Full  (Queue : Queue_Type) return Boolean;

   Queue_overflow, Queue_underflow : exception;

private
   type Marker is mod Queue_Size;
   type List is array (Marker) of Element;
   type Queue_Type is record
      Top, Free : Marker  := Marker'First;
      Is_Empty  : Boolean := True;
      Elements  : List;
   end record;

end Queue_Pack_Private;
```

What do we find in this specification?

- **Constants**, **types**, **procedures** (which are functions without return values), **functions** (which are procedures with return values) and **exceptions**. Functions will gain a more specific meaning in the context of concurrent programming later (as some functions will need to be side-effect free).
- Type definitions can (and should) be highly specific or purposefully undefined. The `Element` type for instance is a new discrete numeric type which covers the range from 1,000 to 24,000. As we defined this as a new type, this type will not combine with other discrete numeric types, like e.g. `Natural`. We could have also expressed that this type should work in combination with another numeric type and could have e.g. stated: `subtype Element is Positive range 1_000 .. 24_000;`

  On the other hand, the `Queue_Type` is purposefully undefined on the user side and even comes with `limited` accessibility, such that the user can not make copies of it, or compare it against another `Queue_Type` instance. When does it make sense to forbid copying?
- The `private` part is not accessible to anybody using this package, but needs to be there so that the specification package can be compiled separately.

Some more observations inside this module which you should take note of:

- Ada allows to initialize everything … and to initialize everything exactly how you want it to be initialized. The most common form (as you see above) is to initialize variables (and constants anyway obviously) on declaration. This also applies to record definitions. You should individually initialize as much as possible, but sometimes it just won't make sense to do so. What Ada does to uninitialized variables is up to you as well: You can leave them uninitialized (meaning they will have random memory contents), you can automatically initialize them to valid values for your type (e.g. `0` for a variable of type `Natural`) or you can ask the compiler to set them automatically to invalid values (like e.g. `-1` for a variable of type `Natural`). The latter is the common form in high-integrity systems where accidental access to uninitialized data is a major disaster and test-suites are supposed to expose such a serious case early. Additionally the compiler will often warn you if there is a chance that data is read before it is written.
- Parameters can be passed as **`in`**, **`out`**, or **`in out`** – which all pose restrictions on the variables or expressions which can be passed. The default is **`in`**, which turns parameters into constants inside those procedures and functions.
- Among the many numerical types is also a modulo type which automatically turns all arithmetic on this type into modulo arithmetic – to the modulo which you specify (this is so-to-speak the useful version of "wrap around arithmetic" as you know it from C, Java or Haskell).
- Types are not for decoration in Ada and so you can (and should) use anything which you have already defined later in your code – avoid double definitions. For instance we use the modulo type `Marker` to define an array over this domain. Right after, we use a type attribute of the type `Marker` (namely `'First`) to initialize a variable of this type in a sensible way.

So far so good, but where is the actual code? Well, here we go in a separate file:

```
package body Queue_Pack_Private is

   procedure Enqueue (Item : Element; Queue : in out Queue_Type) is

   begin
      if Is_Full (Queue) then
         raise Queue_overflow;
      end if;

      Queue.Elements (Queue.Free) := Item;
      Queue.Free     := Queue.Free + 1;
      Queue.Is_Empty := False;
   end Enqueue;

   function Is_Full  (Queue : Queue_Type) return Boolean is
     (not Queue.Is_Empty and then Queue.Top = Queue.Free);

end Queue_Pack_Private;
```

The definitions of `Dequeue` and `Is_Empty` are missing as there is a chance that a mean-spirited lecturer could ask you to actually write those in the next exercise.

What's noteworthy about the implementation code?

- The specifications of procedures and functions are repeated but now followed by a "**`is`** …;" to define their implementation.
- As the type for `Free` is a modulo type which also happens to have been used to define the array, there is no need to check for index bounds as this variable will always stay inside the index range (and will wrap around in the style of a ring buffer, which is exactly what's needed here).
- Sometimes functions can be defined by a single expression. If so, we can skip the **`begin`**-**`end`** bracket as well as the **`return`** statement and just write out the expression to be returned straight away (in parenthesis). Ada offers conditional expressions including **`if`** or **`case`** (with the same meaning as the **`if`** and **`case`** statements, only that in case of an conditional expression there always needs to be a value, i.e. every **`if`** needs to have an **`else`**). Haskell programmers will feel right at home, but will miss the where-clause.

You may ask why we didn't express dequeue as a **function**. In fact we could, but the general notion is that functions are used with **in** parameters only, such that they can be used in side-effect free operations. Depending on convenience, you can break this rule though (since Ada2012) and functions can now also have other parameter passing methods - but not in all circumstances, as some concurrent operations are dependent on side-effect-free operations and hence the compiler will restrict those cases – more on that a little later in the course. If possible just keep your functions pure and you won't get into an argument with the compiler. For those of you who enjoyed functional programming, this will come naturally anyway.

A little food for thought as the concluding aspect of this interlude: If you change anything in a specification or an implementation package what would be the impact on the rest of your software structure? Which part of your program would be dependent on which kind of change? Specifically: what exactly do you need to re-compile? Could this be automated by the compiler or will you need to write a `make` file to express that?

Exercise 2: Make the queue go

Here comes an almost working, very simple program which uses the queue as defined above:

```
with Queue_Pack_Private; use Queue_Pack_Private;
with Ada.Text_IO;        use Ada.Text_IO;

procedure Queue_Test_Private is

   Queue_1,
   Queue_2      : Queue_Type;
   Current_Item : Element;


begin
   Queue_2 := Queue_1;

   Enqueue (Item => 1_000, Queue => Queue_1);

   Dequeue (Current_Item, Queue_1);
   Put_Line ("Current_Item: " & Element'Image (Current_Item));

   Enqueue (Current_Item, Queue_2);

   Dequeue (Current_Item, Queue_1);
   Put_Line ("Current_Item: " & Element'Image (Current_Item));

   Put_Line ("Queue_1 is " &
             (if Is_Empty (Queue_1) then "" else "not ") & "empty on exit");

   Put_Line ("Queue_2 is " &
             (if Is_Empty (Queue_2) then "" else "not ") & "empty on exit");

exception
   when Queue_underflow => Put ("Queue underflow");
   when Queue_overflow  => Put ("Queue overflow");
end Queue_Test_Private;
```

A few comments to help you orient in this program:

- **with** context clauses are used to import items from other packages.
- **use** context clauses implicitly dereference all items inside those packages and allow them to be used directly by their names and without writing <package-identifier>.<item-identifier> for every access. Without them we would for instance need to write `Ada.Text_IO.Put_Line`.
- A top level **procedure** is read as the executable "main" program.
- Variables and types are declared Algol-style, i.e. you can read them from left to right as "`Current_Item` is a variable of type `Element`", as opposed to C/Yoda-style: "Of type `Element` the variable `Current_Item` will be".

- **Parameters** can be passed **by name** (meaning you write down the declared parameter name followed by a `=>` and the parameter you want to pass) or **by order** (you write down all parameters separated by commas and they will be matched in the same order as they were declared for this method) – for obvious mess-avoidance reasons you need to stick with one or the other inside a single parameter list.
- Ada offers **conditional expressions**, here used to potentially add a "not" to the output.
- `exceptions` can be handled in any way you can imagine. Here we just handle two specific exceptions individually (and let all other exceptions slip past). Not catching an exception can be a very bad thing and inside our concurrent systems which we will build in this course, it usually is not a good idea. Yet here on the top level, an uncaught exception will still be displayed by the runtime environment.

A word about the file naming conventions in Ada: All code files need to have lower case names without spaces in them. The file name also needs to match the package or procedure name inside the code file (just as lower-case). Specification files use the extension `.ads` while all other source files use `.adb`.

Download and repair this program such that it compiles and runs without raising an exception. You will also need to fill in the implementations for `Dequeue` and `Is_Empty` to make this go.

If you have it all happily running, zip up the directory again such that you have a new `Queue_Sequential.zip` which follows the exact same structure as the one you downloaded before. Now submit your new zip file for tests and code review by us on the *SubmissionApp* under "Lab 2 Sequential Queue".

Hint: take your time to look around in your development environment (`gps`) and take the hints it gives you while coding. If you keep your code compilable, then the environment can give you significantly more help. Notice for instance how it already suggest to tell you what the parameter list should look like the moment you type an open parenthesis after a function or generic package name. Once you nailed the specification package, the environment can e.g. also produce an outline for the body package (menu: Edit ➥ Generate body). Renaming identifiers throughout the project is one right-click-menu command (obviously requires that the code compiles at this time - otherwise it would not be known what the meanings of all identifiers are). Any feature which you pick up early will make your life junks easier. … and … there is an handy interface to the debugger as well: so set breakpoints, step through your code, display local variables, or – if you enjoyed assembly programming: check out what your program looks like in machine code. If you are up for speed: there are also "Production" and "Performance" scenarios predefined in your project definitions (code optimization and reducing or killing all run-time checks and debugging hooks).

Interlude: Generic programming

You know the idea from your polymorphic types in Haskell and from your templates in Java or C++. As in Haskell, generic packages need to be locally compilable without any knowledge of any concrete types which will be used later to instantiate it. This is different from the template concept in e.g. C++, where the full compilation tests are performed separately for every instance of a template. While this sounds all tremendously complicated it is actually as easy as in Haskell:

Remove the line where the Element type is defined in the specification package and add:

```
generic
   type Element is private;
```

in front of the package definition. Voilà: you just wrote your first generic package in Ada, where the `Element` type can be instantiated with anything. The compiler will make sure that your generic package compiles on its own. In this case the compiler will check that no operation besides copying or comparing is ever applied to variables of type `Element`, as this generic defini-

tion states that Element can be anything (which checks out for our queue package, as we never actually do anything with the elements of the queue besides copying them in and out).

Alternatively you could (even though it would be silly here), restrict the Element type e.g. to of a certain numeric type, or to have certain operations defined on it. The feature of specifying exactly which operations need to exist for a specific type gives you all options to be highly specific (without the need to define a new type-class, like in Haskell). If you need order on the type you can for instance define:

```
generic
   type Element is private;
   with function "<" (left, right : Element) return Boolean is <>;
```

which restricts the Element type to types for which < is defined. As we said, this was just an example and it doesn't make sense for our queue package (in fact the compiler will point out that this function is an unnecessary restriction), but it would for example make sense for a package which sorts generic elements.

While we are in the polymorphism section: You can also overload anything in Ada, including default operations like "=", "<" or "*". Type class restrictions as in Haskell do not apply here. The only obvious constraint is that at least one parameter type needs to be different (this can also be the return type of a function).

If I now assume that we called our new generic package `Queue_Pack_Generic` then we can produce concrete instances of this generic package like this:

```
with Ada.Text_IO;          use Ada.Text_IO;
with Queue_Pack_Generic;

procedure Queue_Test_Generic is

   package Queue_Pack_Character is new Queue_Pack_Generic (Element => Character);
   use Queue_Pack_Character;

   Queue : Queue_Type;
   Current_Item  : Character;

begin
   Enqueue (Item => 'x', Queue => Queue);
   Enqueue (Item => 'y', Queue => Queue);
   Enqueue (Item => 'z', Queue => Queue);

   Dequeue (Current_Item, Queue);
   Put_Line ("Current_Item: " & Current_Item);

   Put_Line ("Queue is " &
                    (if Is_Empty (Queue) then "" else "not ") & "empty on exit");
exception
   when Queueunderflow => Put ("Queue underflow");
   when Queueoverflow  => Put ("Queue overflow");
end Queue_Test_Generic;
```

Note that the `use` clause can only be applied to instances of our generic package – not to the generic package itself. Package instantiations are part of a declare block. After a package instantiation, everything else works exactly as before. We can now instantiate queues for any type we wish and also multiple times inside the same declare block.

Exercise 3: Stacks

Now produce a generic Stack package (reminder: stacks are First-In-Last-Out as opposed to queues which are First-In-First-Out storage structures) and a test program to demonstrate its sanity. While you are at it, make the stack size generic as well and precede your package specification with:

```
generic
   type Element is private;
   Stack_Size : Positive := 10;
```

To make your life easier: make a copy of the `Queue_Sequential` directory which you have from the last exercise and call it `Stack_Sequential`, enter this new directory, call `gps&` and change the name of the project to Stack_Sequential (in menu Project ➥ Edit Project Properties). Finally remove the left-over project file `queue_sequential.gpr` in your new directory. You can follow the structure of your queue module to a large degree, but the semantic will need to change with respect to the actual operations. Think how you will represent an empty or a full stack.

Try to be as specific and precise as you can and submit your result for code review by us on the *SubmissionApp* under "Lab 2 Generic Stacks". We are serious about details (as any professional should be) regardless how complex or trivial the algorithm.

Exercise 4: Hold instead of break

Advanced, race-car-red exercises are not for the faint-of-heart, but rather for the swift, the brave and the foolish. Do them for a more solid understanding of the field or for exercise later when you revisit your earlier labs. If you are targeting a High Distinction mark in this course, I also strongly recommend that you complete all advanced exercises.

The advanced exercises frequently contain methods to solve some of the central problems of the current lab in a more elegant, yet also more demanding way.

Here comes your first excursion into concurrency territory and you will find that your code can become shorter and easier by thinking concurrently. So far we have those ugly exceptions in our code as there is no nice way out, if our sequential program attempts to read out a value from a queue where there is none.

In the concurrent world, there is no need to press the panic button quite yet: while there might not be a value in the queue right now, there could will be one in the future as other tasks are still running all around our temporarily disappointed task. So why not wait for a bit until some other friendly task left a new element in the queue and then we continue smoothly?

This is achieved by making a call to the queue conditional. Concretely: we are only allowed to read a value if the queue is currently not empty. Respectively we are only allowed to write a value if the queue is currently not full. In all other cases we make the calling task wait for a bit (in fact we queue it up on this call) until the condition changes. An entity which can hold one or multiple calling tasks is called a **protected entry** and the condition is called a **guard**.

So the ugly sequential code:

```
procedure Enqueue (Item : Element; Queue : in out Queue_Type) is
   begin
      if Is_Full (Queue) then
         raise Queueoverflow;
      end if;
      …
```

changes into the concurrent code:

```
entry Enqueue (Item : Element) when not Is_Full is …
```

Such an entry will need to be a part of a **protected object** as it protects a data structure (here: our Queue) against ill-posed concurrent access forms (as you will find in your next lab). This means our access methods will all become part of a protected object as in:

```
pragma Initialize_Scalars;
generic
   type Element is private;
   type Index   is mod <>;  -- Modulo defines size of the queue.
package Queue_Pack_Protected_Generic is
   type Queue_Type is limited private;
   protected type Protected_Queue is
      entry Enqueue (Item :     Element);
      entry Dequeue (Item : out Element);
      function Is_Empty return Boolean;
      function Is_Full  return Boolean;
   private
      Queue : Queue_Type;
   end Protected_Queue;
private
   type List is array (Index) of Element;
   type Queue_Type is record
      Top, Free : Index   := Index'First;
      Is_Empty  : Boolean := True;
      Elements  : List;
   end record;
end Queue_Pack_Protected_Generic;
```

while the implementation will now become much nicer:

```
package body Queue_Pack_Protected_Generic is
   protected body Protected_Queue is
      entry Enqueue (Item : Element) when not Is_Full is
      begin
         Queue.Elements (Queue.Free) := Item;
         Queue.Free     := Index'Succ (Queue.Free);
         Queue.Is_Empty := False;
      end Enqueue;
      entry Dequeue (Item : out Element) when not Is_Empty is
      begin
         Item           := Queue.Elements (Queue.Top);
         Queue.Top      := Index'Succ (Queue.Top);
         Queue.Is_Empty := Queue.Top = Queue.Free;
      end Dequeue;
      function Is_Empty return Boolean is (Queue.Is_Empty);
      function Is_Full  return Boolean is
        (not Queue.Is_Empty and then Queue.Top = Queue.Free);
   end Protected_Queue;
end Queue_Pack_Protected_Generic;
```

Instead of one sequential program to use our queue, we can now have e.g. three concurrent entities (tasks) which will read and three which will write on this queue:

```
with Ada.Task_Identification;      use Ada.Task_Identification;
with Ada.Text_IO;                  use Ada.Text_IO;
with Queue_Pack_Protected_Generic;

procedure Queue_Test_Protected_Generic is

   type Queue_Size is mod 3;

   package Queue_Pack_Protected_Character is
      new Queue_Pack_Protected_Generic (Character, Queue_Size);
   use Queue_Pack_Protected_Character;

   subtype Some_Characters is Character range 'a' .. 'f';

   Queue : Protected_Queue;

   type Task_Index is range 1 .. 3;

   task type Producer;
   task type Consumer;

   Producers : array (Task_Index) of Producer;
   Consumers : array (Task_Index) of Consumer;

   task body Producer is

   begin
      for Ch in Some_Characters loop
         Put_Line ("Task " & Image (Current_Task) & " finds the queue to be " &
                   (if Queue.Is_Empty then "EMPTY" else "not empty") &
                     " and " &
                   (if Queue.Is_Full  then "FULL" else "not full") &
                     " and prepares to add: " & Character'Image (Ch) &
                     " to the queue.");
         Queue.Enqueue (Ch); -- task might be blocked here!
      end loop;
      Put_Line ("<---- Task " & Image (Current_Task) & " terminates.");
   end Producer;

   task body Consumer is

      Item    : Character;
      Counter : Natural := 0;

   begin
      loop
         Queue.Dequeue (Item); -- task might be blocked here!
         Counter := Natural'Succ (Counter);
         Put_Line ("Task " & Image (Current_Task) &
                     " received: " & Character'Image (Item) &
                     " and the queue appears to be " &
                   (if Queue.Is_Empty then "EMPTY" else "not empty") &
                     " and " &
                   (if Queue.Is_Full  then "FULL" else "not full") &
                     " afterwards.");
         exit when Item = Some_Characters'Last;
      end loop;
      Put_Line ("<---- Task " & Image (Current_Task) &
                " terminates and received" & Natural'Image (Counter) & " items.");
   end Consumer;

begin
   null;
end Queue_Test_Protected_Generic;
```

Your job for this last exercise is to run this code and to make heads and tails out of the output which you gain. A piece of paper with three producers and three consumers on it with the

queue in the middle and your pencil adding and removing characters from this queue might come in very handy. Specifically you should answer the following questions:

- Why is it that you might observe that tasks just added something to the list and find the list empty straight after? Conversely: why is that tasks just read something from the list and still find the list full to the brim right after?
- Would it be possible that a tasks states in the above program that the queue is empty *and* full?
- Will all producers write the same number of elements?
- Will all consumers read out the same number of elements?
- Will the total number of elements written equal the total number of elements read?
- Did we get rid of all exceptions for good? Did we introduced a new, potentially troublesome issue by our "hold instead of break" method?

Submit your answers on the *SubmissionApp* under "Lab 2 Observations". Just type your answers directly into the submission box there. No need for more than a sentence per question. Always give a reason, so write for instance something like: "Never, because …"

I am well aware that you need to guess some of the syntax at the moment, but you should be able to interpret some of the output which you gain and to draw some early conclusions about what concurrent systems can do. This is the point of this last, first week's exercise.

Make Sure You Logout to Terminate Your Session!

Outlook

Next week you will be creating your own first tasks and will see how easy the first steps towards concurrent systems are … with a little example how things can also go bad at the end.