Part I. Multiple Choice and Short Answer. Each question is worth 3 points.

When building a software system, you can choose to use Abstract Data Types (ADTs) or you can build the necessary functions into the software on an ad hoc basis. For questions 1 and 2 choose the one of the two situations where you would be more likely to choose to use ADTs.

**B** 1. A. All developers know how all the software works.  
    B. Not all developers need to know how all the software works.

**A** 2. A. The software being developed is intended to be part of other systems.  
    B. The software being developed will only be used in this particular system.

**B** 3. A type where equality ("=") and assignment (":=") are the only operations that are available is a(n):  
A. Ordinary TYPE  
B. PRIVATE TYPE  
C. PRIVATE DETECTIVE TYPE  
D. LIMITED PRIVATE TYPE

For each of the following answer T (True) if it can be generic and F (False) if it cannot be generic. For example, for the first question answer T if you can have a GENERIC PROCEDURE and F if you cannot have a GENERIC PROCEDURE.

**T** 4. PROCEDURE  
**F** 5. TYPE  
**T** 6. PACKAGE  
**F** 7. TASK

For question 8 to 11 Answer T (True) if the following are linear data structures, answer F (False) if they are not.

**F** 8. RECORD  
**T** 9. One dimensional ARRAY  
**F** 10. ACCESS Type  
**T** 11. Queue
For each of the following answer T (True) or F (False):

__F__ 12. You can create instances of ABSTRACT Types.

__T__ 13. Primitive Subprograms can be ABSTRACT.

__F__ 14. Non-primitive Subprograms can be ABSTRACT.

__T__ 15. A concrete (non ABSTRACT) TAGGED TYPE can have a primitive subprogram.

__F__ 16. Any subprogram which has an argument of a TAGGED TYPE can be dynamically dispatched.

__T__ 17. A primitive subprogram for a TAGGED TYPE can be dynamically dispatched.

__F__ 18. When a primitive procedure of a TAGGED TYPE is called, the most general (non-specific) version of the procedure is the one that is called.

__T__ 19. All TAGGED Types have a TAG field added by the compiler as part of their record structure.

For question 20 to 24, consider the following code fragment:

```pascal
package P is
  type T1 is tagged record ... end record;
  procedure P1( X : in T1 );
end P;

X1 : P.T1'Class;
X2 : P.T1'Class := ... ;

type T1Ptr is access P.T1'Class;
procedure P2( X : in P.T1'Class ) is ... end P2;
```

Answer T (True) or F (False) for each of the following:

__T__ 20. P1 is a Primitive Subprogram of T1.

__F__ 21. The declaration of X1 is legal.

__T__ 22. The declaration of X2 is legal.

__T__ 23. The declaration of T1Ptr is legal;

__F__ 24. P2 is a Primitive Subprogram of T1.
Ada TASKing is an example of heavy weight multi-threading.

Ada TASKing is based on Cooperating Sequential Processes and Rendezvous.

For question 27 to 31 answer T (True) if the concept is something that is appropriate to use in a program using tasking. Answer F (False) if the concept is something you don't want (e.g. is probably a bug) in a program using tasking.

27. Race conditions 
F

28. Protected Objects 
T

29. Deadlock 
F

30. Timing Holes 
F

31. Producer/Consumer tasks 
T

For questions 32 and 33, fill in the requested numbers.

Consider the following program:

```ada
with Ada.Integer_Text_IO;
procedure DoIt is
  type Integer_Array is array(Positive range <>) of Integer;
  function Recursive( IA : Integer_Array ) return Integer is
    begin
      if IA'Length = 0 then
        return 0;
      else
        return IA( IA'First )
          + Recursive( IA( IA'First + 1 .. IA'Last ));
      end if;
    end Recursive;

  X : Integer_Array := ( 1, 2, 3, 4);
  begin
    Ada.Integer_Text_IO.Put( Recursive(X) );
    end DoIt;
end DoIt;
```

32. What value is printed by the program? 

10

33. How many times is Recursive called in order to print the value given in the previous question.

5
Part 2. Programming problem. Write your answers in the space provided. Use the back of the page if necessary. Each question is worth 10 points. All problems in this part concern the following "really simple bounded Integer stack" ADT:

```plaintext
package A_Really_Simple_Integer_Stack is
    type Integer_Array is array(Positive range <>) of Integer;
    Default_Array_Size : constant Integer := 5;

    type Stack is record
        Size : Natural := 0;
        Data : Integer_Array( 1 .. Default_Array_Size);
    end record;

    function Top(S : Stack) return Integer;
    -- Return the entry on top of the stack

    procedure Push(S : in out Stack; V : in Integer);
    -- Push the value V onto the stack S

    procedure Pop(S : in out Stack);
    -- Pop the value off the top of the stack
end A_Really_Simple_Integer_Stack;

package body A_Really_Simple_Integer_Stack is
    function Top(S : Stack) return Integer is
    begin
        return S.Data(S.Size);
    end Top;

    procedure Push(S : in out Stack; V : in Integer) is
    begin
        S.Size := S.Size + 1;
        S.Data(S.Size) := V;
    end Push;

    procedure Pop(S : in out Stack) is
    begin
        S.Size := S.Size - 1;
    end Pop;
end A_Really_Simple_Integer_Stack;

Note: Don't be concerned that this implementation ignores all the obvious exceptions that might be raised for stack full or empty. A Constraint_Error will be raised if any of the obvious restrictions are violated.

The stack above is a bounded stack. That is, there is a limit on the number of Integers that can be pushed onto the stack. However, what is desired is a dynamic stack: a stack that grows as needed to hold the data pushed on the stack. We are going to modify the above package to implement a dynamic stack.

Suggestion: Read all the questions before starting to answer them. Hint: The obvious answer is, in many cases, the right answer.
1. First we need to modify our data structure. We need to modify it to use a pointer to an ARRAY rather than an ARRAY. Write the declaration for a pointer type to Integer_Array and indicate where in the above code you would place the declaration.

   ```ada
   type Integer_Array_Ref is access Integer_Array;
   -- After declaration of Integer_Array, see (1), above
   ```

2. Second, write a new version of the declaration for the Stack type, changing Data to be a pointer to an Integer_Array. Make sure to provide appropriate initial values for the fields in your new record.

   ```ada
   type Stack is record
       Size : Natural := 0;
       Data : Integer_Array_Ref :=
           new Integer_Array(1 .. Default_Array_Size);
   end record;
   ```

3. Even though the length of the array pointed to by your new Stack type will be variable, your new record does not need to keep track of the length of the array. Why not?

   Because arrays in Ada keep track of their length. That is, you can always get the length of the array as `Data.all'Length`.

4. If any modification is required to the body of the procedure Top provide it below, or state No Modification required.

   No modification required.
   Or, change `return S.Data(S.Size)` to `return S.Data.all(S.Size)`

5. If any modification is required to the body of the procedure Pop provide it below or state No Modification required.

   No modification required.
The procedure push will require some significant modification. If more data is pushed than the array will hold, we need to create a new, larger array and get rid of the old one. First, since we are discarding the old one, we need an instance of Unchecked_Deallocation for the Integer_Array pointer type previously declared.

6. Write an instantiation of Unchecked_Deallocation to be used for our new stack.

```plaintext
procedure Deallocate_Array is
new Unchecked_Deallocation(
  Object => Integer_Array,
  Name   => Integer_Array_Ref);
```

7. Finally write a new version of the Push procedure. It needs to do the following:

   a. Check to see if the already allocated array will hold the data. If it will, skip to step (e).
   b. Allocate a new array that is Default_Array_Size elements larger than the current array.
   c. Copy the values from the old array into the new array.
   d. Deallocate the old array.
   e. Push the value V onto the array.

```plaintext
procedure Push( S : in out Stack; V : Integer) is
  temp : Integer_Array_Ref;
begin
  -- (a) Check to see if array will hold data
  if S.Size = S.Data.all'Length then
    -- Array won't hold the new data, save old array
    temp := S.Data;
    -- (b) Allocate a new array
    S.Data := new Integer_Array(
      1 .. temp.all'Length + Default_Array_Size);
    -- (c) Copy values from old array to new array
    S.Data.all(1 .. temp.all'Length) := temp.all;
    -- (d) Deallocate old array
    Deallocate_Array(temp);
  end if;
  -- (e) Push the value onto the array
  S.Size := S.Size + 1;
  S.Data.all(S.Size) := V;
end Push;
```
8. Now that we have this implementation we want to use it in a program with multiple tasks. Unfortunately, this implementation can fail badly if it is used by multiple tasks "simultaneously". Give one example of the kind of things that could go wrong.

There are many possible answers to this question. Here's one:

If two separate tasks try to Push something on the stack "simultaneously", it is possible that both tasks will increment S.Size, resulting in S.Size being incremented once, not twice. This example is similar to the example of changing account balances given in lecture.

9. Suggest an approach (one or two sentences) for how you might go about fixing the problems noted in the previous problem.

One possible answer:

Make Stack into a protected type.

10. Our implementation only handles stacks of Integers. We would like a generic implementation that would work for any PRIVATE type. Explain why our implementation cannot be used with a LIMITED PRIVATE type.

We copy values (=) into the stack in Push.

Note: In addition, Top is at best problematic. Top would, in fact, be legal with a limited private type. However, the caller would be limited in what could be done with the value returned by Top.