DEFINITION MODULE genericstack;

(* This module defines the public interface for the generic stack abstract data type. It imports the type WORD, which permits the type of elements to be stored in the stack to be bound to the type ARRAY OF WORD, thereby permitting generic elements to be stored in the stack. *)

FROM SYSTEM IMPORT
   (* type *) WORD;

EXPORT QUALIFIED
   (* type *) stack,
   (* proc *) define, makeempty, empty, push, pop;

TYPE stack;

PROCEDURE define
   ( VAR s : stack (* out *) );
(* Creates an empty stack. Must be used before any other stack operation. *)

PROCEDURE makeempty
   ( VAR s : stack (* in/out *) );
(* Reinitializes an existing stack s to an empty stack by removing all elements contained in the stack. *)

PROCEDURE empty
   ( s : stack (* in *) );
(* Returns true if the stack s contains no elements, otherwise returns false. *)

PROCEDURE push
   ( VAR s : stack (* in/out *) ;
      item : ARRAY OF WORD (* in *) );
(* Adds item to the top of stack s. *)

PROCEDURE pop
   ( VAR s : stack (* in/out *) ;
      VAR item : ARRAY OF WORD (* out *) );
(* Removes item from the top of stack s. *)

END generic stack.
Notice that our interface to the generic stack ADT is identical to the interface presented earlier except that the operation "topofstack" is not included. Any program units that use the earlier versions of the stack ADT can use this version without modification, provided they do not use the "topofstack" operation.

Why have we excluded the "topofstack" operation? To be compatible with the earlier versions of the stack ADT, the interface to the generic version would have to have the form:

```plaintext
PROCEDURE topofstack( s : stack (* in *) ) : ARRAY OF WORD;
```

This statement is syntactically incorrect, since the type `ARRAY OF WORD` is not bound to any type. However, we can avoid this problem by defining this operation to be a procedure rather than a function with the following interface:

```plaintext
PROCEDURE topofstack( s : stack (* in *); 
VAR item : ARRAY OF WORD (* out *) );
```

In this case, the type `ARRAY OF WORD` is bound to the type of item. We leave it as an exercise to implement the generic version of this operation.

Now let us work through the implementation details by considering several options for the representation of the stack opaque type. Since we do not know in advance the size of the items the stack will contain, we cannot define fixed size nodes for the items. Our solution is to define nodes that contain two fields: a pointer to the next stack node and the address of the item. We will then use the `ALLOCATE` operation to allocate storage at the address to accommodate the item. The following declarations seem appropriate:

```plaintext
TYPE
  stackptr = POINTER TO stacknode;

  stacknode = RECORD
    contents : ADDRESS
    next     : stackptr;
    END (* record *);

  stack     = stackptr;
```

Conceptually, a generic stack containing three items is depicted in Figure 2.15
Notice that this representation permits items of different size, and more likely of different types, to be placed in the same stack. It is not clear that many applications exist that require a stack with this much generality. Such generality also puts an extra burden on the user of the generic stack. For example, a “pop” operation must bind the returned type ARRAY OF WORD to a variable of a type defined in the user's program. If the returned item is not compatible with the size of the variable in the user's program, an execution error will occur.

We propose to make the generic stack more restrictive by requiring that all elements in the stack be of the same size. The user of the generic stack is not yet completely protected, since the elements can still be of different types as long as they are the same size. We do not know the size of the elements to be used with a given stack variable until we “push” the first item onto the stack. We need to save this size information to verify that later “push” operations are valid and to implement the “pop” operation.

These considerations suggest that we associate with each stack a header node that contains the size information along with a pointer to the rest of the stack. We propose the following declarations:

```plaintext
TYPE
  stack    = POINTER TO stackheader;
  stackptr = POINTER TO stacknode;
  stackheader = RECORD
    size : CARDINAL;
    next : stackptr;
  END (* record *);
  stacknode  = RECORD
    contents : ADDRESS;
    next    : stackptr;
  END (* record *);
```

Figure 2.16 illustrates a generic stack containing three items of two words each.
We now consider the implementation details for the preceding declarations. The define operation must initialize the header node by assigning the size field a value of zero and the next field a value of NIL. We adopt the convention that the stack is empty if the next field of the header node is NIL.

Let us examine the “push” and “pop” operations in detail by presenting pseudo-code descriptions. The implementation details for the “makeempty” operation should then be apparent.

Procedure push ( s : stack; item : array of word )
    Allocate a new stack node
    Determine the size of item
    If the size field of the header node is zero
        Then
            Set the size field of the header node, since this is the first element on the stack
        Else
            If the size of the item is not the same as the size field of the header node
                Then
                    Print an error message and halt execution of the program
                End If
            End If
    End If
    Allocate storage to accommodate the item
    Transfer the item to the allocated storage
    Set the next field of the new stack node to the value of the next field of the header node
    Set the next field of the header node to point to the new stack node
End push.

Procedure pop ( s : stack; item : array of word )
    If the stack is empty
        Then
            stack underflow
        Else
            Get the size of the elements from the header node
            Set a pointer to the top stack node
            Get the memory location of the item from the address field of the top stack node
End Else.
Transfer the contents of size successive memory locations to the item
Dealocate the memory space that contained the item
Set the next field of the header node to the next field of the top of stack node
Dealocate the top of stack node

End if
End pop.

Listing 2.11 presents the implementation details for the generic stack ADT. The reader should have little trouble understanding the implementation, which closely follows the pseudo-code algorithms presented above.

Listing 2.11:

IMPLEMENTATION MODULE genericstack;

FROM InOut IMPORT
  (* proc *) WriteLn, WriteString, WriteCard;

FROM SYSTEM IMPORT
  (* type *) WORD, ADDRESS,
  (* proc *) TSIZE;

FROM Storage IMPORT
  (* proc *) ALLOCATE, DEALLOCATE;

TYPE
  stack = POINTER TO stackheader;
  stackptr = POINTER TO stacknode;

  stackheader = RECORD
    size : CARDINAL;
    next : stackptr;
  END (* record *)

  stacknode = RECORD
    contents : ADDRESS;
    next : stackptr;
  END (* record *)

PROCEDURE define
  ( VAR s : stack (* out * ) )
BEGIN
  NEW( s )
  s^.size := 0;
  s^.next := NIL;

PROCEDURE makeempty
  ( VAR s : stack (* in/out *) );

VAR
  node1 : stackptr;
  node2 : stackptr;
  size : CARDINAL;

BEGIN
  size := s^.size;
  node1 := s^.next;
  WHILE node1 <> NIL DO
    node2 := node1;
    DEALLOCATE( node1^.contents, size );
    node1 := node1^.next;
    DISPOSE( node2 );
  END (* while loop *);
  s^.size := 0;
  s^.next := NIL;
END makeempty;

PROCEDURE empty
  ( s : stack (* in *) ) : BOOLEAN;

BEGIN
  RETURN s^.next = NIL;
END empty;

PROCEDURE push
  ( VAR s : stack (* in/out *);
    item : ARRAY OF WORD (* in * ) );

VAR
  size : CARDINAL;
  newnode : stackptr;
  wordcount : CARDINAL;

BEGIN
  NEW( newnode );
  (* Calculate size of item in bytes. *)
  size := ( HIGH( item ) + 1 ) * TSIZE( WORD );
  IF s^.size = 0 (* This is first item on stack. *)
    THEN (* Set size of items in header node. *)
      s^.size := size;
  ELSIF s^.size # size (*This is not the 1st item on stack*)
    THEN (* The size of item is not compatible. *)
      WriteLn;
WriteString( "Error Attempting to push an object of");
WriteString( " inconsistent size onto stack." );
HALT;
END (* if then * )
ALLOCATE( newnode^.contents, size);
location := newnode^.contents;
FOR wordcount := 0 TO HIGH( item ) DO
  location^ := item[ wordcount ];
  INC( location, TSIZE( WORD ) );
END (* for loop * )
newnode^.next := s^.next;
s^.next := newnode;
END push;

PROCEDURE stackunderflow;
(* Error handling procedure: message, recovery, abort. *)
BEGIN
  WriteLn; WriteLn;
  WriteString( "Error attempting to pop and empty stack." );
  WriteLn;
  HALT;
END stackunderflow;

PROCEDURE pop
( VAR s : stack (* in/out * ));
  VAR item : ARRAY OF WORD (* out * ));

VAR
  size : CARDINAL;
  oldnode : stackptr;
  wordcount : CARDINAL;
  location : ADDRESS;

BEGIN
  IF empty( s )
  THEN
    ELSE
      size := s^.size;
      oldnode := s^.next;
      location := oldnode^.contents;
      FOR wordcount := 0 TO size DIV TSIZE( WORD ) - 1 DO
        item[ wordcount ] := location^;
        INC( location, TSIZE( WORD ) );
      END (* for loop * );
      DEALLOCATE( oldnode^.contents, size );
      s^.next := oldnode^.next;
      DISPOSE( oldnode );
    END (* if then * )
  END pop;

END genericstack.