Revisiting extensions to Modula−2 to improve reusability

Jürgen Börstler and Albert Zündorf
RWTH Aachen, Lehrstuhl für Informatik III,
Ahornstraße 55, D−5100 Aachen,
Germany

1. INTRODUCTION

Achieving reusability is a central goal in programming. Since code can seldom be reused "as−is", it is advantageous to have language concepts that help in adapting code to different needs. In modern programming languages reuse is supported by constructs for encapsulation, genericity, and inheritance. But most programming languages in use do not support all of these facilities. It is therefore interesting to study how these languages can be used or maybe extended to support reuse.

We have chosen Modula−2 for this discussion for several reasons: (a) the recent discussion about this topic in the literature, (b) our own experiences, and (c) the simplicity of the language. It is our persuasion that most of the discussion can simply be transferred to other (procedural) languages.

In this paper we review solutions for Modula−2 proposed in the literature. In addition we also present our own variations and extensions to these solutions. We classify these solutions into solutions without changing the language, and solutions which extend Modula−2 in some way. Since Modula−2 is a strongly typed language without explicitly supporting genericity and/or inheritance, it is difficult to write really reusable code (e.g. generic abstract data types). Therefore, Modula−2 contains constructs for by−passing the type checking (`ADR(...)`, `ADDRESS`, `ARRAY OF BYTE/WORD`). These constructs can be used to introduce adaptability.

The first part of the paper reviews Modula−2 with respect to the definition of generic abstract data types (ADTs). The second part of this paper discusses object oriented extensions of Modula−2 and how they can be combined with the generic extensions proposed in the first part of the paper.

Several examples are presented to clarify our ideas.

2. GENERICITY IN MODULA−2

In this chapter we discuss several approaches to define generic ADTs in Modula−2. Generic ADTs are very useful, because a lot of ADTs, like stacks, lists, trees, and graphs, etc. may be defined independently of the type of the elements (called base types) stored in the structure.

2.1. Solutions without changing the language

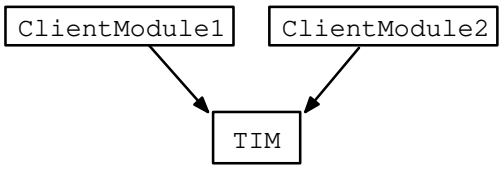
Following /LeSp 87/ in our technical terms there are three different methods to introduce genericity: The type independent module (TIM), and the element module (EM) method proposed by /WiSi 85/, and the intermediate module (IM) method proposed by /CzIg 85/. Furthermore, there is an approach proposed by /Lins 89/, which we will call type manager module

(TMM) method.

A TIM is the implementation of an generic ADT, where all operations are implemented using formal parameters of type `ARRAY OF BYTE`, since this formal parameter type is compatible with actual parameters of any type[1]. So client modules can use the operations provided by the TIM without paying attention to the base type(s) of the generic ADT. However, overriding the type checking facilities can lead to serious run−time failures, which are difficult to discover (see (*1*) in example 2.1). The advantages of the TIM method are its simplicity and space efficiency. It requires no redundant code and there is no need to recompile the TIM when base types are added or changed.

Example 2.1: The type independent module (TIM) method

```
DEFINITION MODULE TIM_Stack;
   FROM SYSTEM IMPORT BYTE;
   TYPE Stack;         (* the generic type *)
   PROCEDURE Create( VAR S: Stack);
   PROCEDURE Push( VAR S: Stack; data: ARRAY OF BYTE);
   ROCEDURE Pop( VAR S: Stack; VAR data: ARRAY OF BYTE);
END TIM_Stack.
```




```
MODULE ClientModule;
   FROM TIM_Stack IMPORT
      Stack, Create, Push, Pop;
   VAR S1, S2: Stack;
       i: INTEGER;     (* a base type *)
       c: CHAR;        (* another base type *)
BEGIN
   (* ... *)
   Create( S1);
   Create( S2);
   Push( S1, i);       (* use 'S1' as stack for integers *)
   Push( S2, c);       (* use 'S2' as stack for characters *)
   S1 := S2;           (*1*)
   (* ... *)           (* operations on 'S1' will cause problems *)
END ClientModule.
```

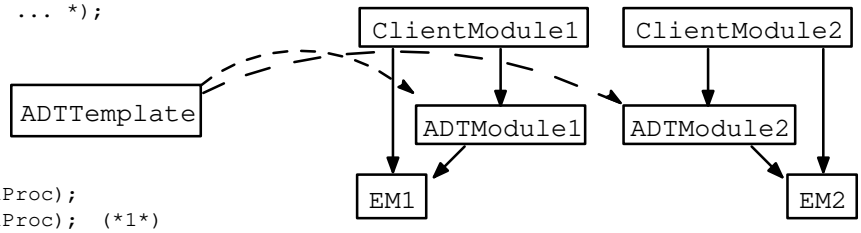
In the EM method all operations of the ADT require some `ElementType` as formal parameter, which has to be imported from the EM. For each new or changed base type clients must provide an EM exporting that base type. In addition they must also create an almost redundant ADT module importing this base type. Some of this work can be automated by a simple preprocessor duplicating (a template of) the ADT module and substituting the new base type for `ElementType`. The main disadvantage of this method is the need to create a lot of redundant code for each new instance of the generic ADT. This also can lead to serious maintenance problems, if the implementation of the ADT has to be changed. This method only supports homogeneous data structures, i.e. they can only hold one basetype at the same time (see (*1*) in example 2.2). Opposed to the TIM method this method enforces strong type checking.

Example 2.2: The element module (EM) method

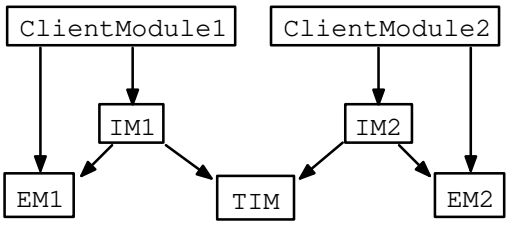
```
DEFINITION MODULE ADTTemplate_Set;
   FROM EM IMPORT ElementType;    (* the formal generic type parameter *)
   TYPE EqualProc = PROCEDURE( ElementType, ElementType): BOOLEAN;
        (* the constraints the actual generic type parameter must fulfil *)
   TYPE Set;                     (* the generic type *)
   PROCEDURE Create( VAR S: Set);
   PROCEDURE Insert( VAR S: Set; data: ElementType; Equal: EqualProc);
   (* ... *)
END ADTTemplate_Set.
DEFINITION MODULE EM1;
```

1. *Originally the formal parameters should be of type* `ARRAY OF WORD` *(/Wirth 85/), but on byte addressable machines* `ARRAY OF BYTE` *can be used.*

```
   TYPE SomeElementType;
   PROCEDURE SomeEqualProc( e1, e2: SomeElementType): BOOLEAN;
   (* ... *)
END EM1.
DEFINITION MODULE ADTModule1_Set;
   FROM EM1 IMPORT SomeElementType;          (* the actual generic type parameter *)
   TYPE EqualProc = PROCEDURE( ElementType, ElementType): BOOLEAN;
        (* the constraints the actual generic type parameter must fulfil *)
   TYPE Set;
   PROCEDURE Create( VAR S: Set);
   PROCEDURE Insert( VAR S: Set; data: SomeElementType; Equal: EqualProc);
   (* ... *)
END ADTModule_Set.
MODULE ClientModule1;
   FROM ADTModule1_Set IMPORT
      Set, Create, Insert (* ... *);
   FROM EM1 IMPORT
      SomeElementType,
      SomeEqualProc (* must conform to signature given by ADTTemplate_Set.EqualProc *)
      (* ... *);
   TYPE SomeOtherType = (* ... *);
   VAR S: Set;
       e: SomeElementType;
       o: SomeOtherType;
BEGIN
   (* ... *)
   Create( S);
   Insert( S, e, SomeEqualProc);
   Insert( S, o, SomeEqualProc);  (*1*)
   (* ... *)                      (* compiler will detect an error here *)
END ClientModule1.
```



The IM method is a combination of the EM and the TIM method. Each IM provides client modules with an ADT of one base type. It therefore imports this base type from an EM and the ADT interface from a TIM. This method is very similar to the EM method, but significantly reduces code redundancy by using one TIM on which the IMs are built upon to enforce type checking. To save space we omit an example for this method.



The TMM method as proposed by /Lins 89/ is a further development based on ideas proposed by /Goug 85/ and /BrHa 86/. This method only allows generic ADTs with base types (called `items`) compatible to `ADDRESS`. The basic definitions of such base types are encapsulated in a module `Items`, which provides an abstract interface for generic items. To ensure that base types conform to the type `Item` defined in `Items`, each new base type is registered and validated by the TMM. The TMM ensures that this new item type receives an unambiguous `TypeID`. In addition the TMM associates operations to this new type for assigning, comparing, and disposing items of that type[2]. These operations have to be supported by the clients at instantiation time of a new item type. The `TypeID` can then be used as base type to instantiate the generic ADT. This method ensures strong typing and avoids the duplication of code. Since the operations to assign, compare, and dispose items of a given type are managed by the TMM there is a little overhead in calling these operations. In addition there is some extra effort for the clients in providing operations for type transfer and conversions.

2. For the "standard" types CHAR, CARDINAL, INTEGER, REAL, LONGCARD, and LONGINT these instantiations are predefined.

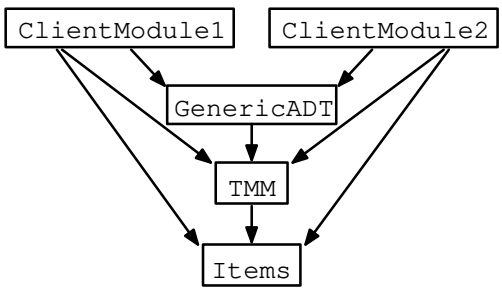
Example 2.3: The type manager module (TMM) method

```
DEFINITION MODULE Items;
   FROM Relations IMPORT Relation (* = (less, equal, ...) *);
   TYPE Item (* = ADDRESS *);
   (* ... *)
   TYPE AssignProc = PROCEDURE( Item): Item;
        DisposeProc = PROCEDURE( VAR Item);
        CompareProc = PROCEDURE( Item, Item): Relation;
   (* ... *)
END Items.

DEFINITION MODULE TMM;
   FROM Items IMPORT AssignProc, CompareProc, DisposeProc;
   TYPE TypeID;
   TYPE TypeName = ARRAY [0..15] OF CHAR;
   (* ... *)
   PROCEDURE TCreate( the Name: TypeName;
                      assignment: AssignProc;
                      comparison: CompareProc;
                      deallocation: DisposeProc): TypeID;
   (* ... *)
END TMM.

DEFINITION MODULE GenericADT_Set;
   FROM Items IMPORT Item;
   FROM TMM IMPORT TypeID;
   TYPE Set;
   PROCEDURE Create( theType: TypeID): Set
   PROCEDURE Insert( VAR theSet: Set; data: Item);
   (* ... *)
END GenericADT_Set.

MODULE ClientModule1;
   FROM Items IMPORT AssignProc, CompareProc, DisposeProc;
   FROM TMM IMPORT TypeID, TCreate (* ... *);
   FROM GenericADT_Set IMPORT Set, Create, Insert (* ... *);
   TYPE SomeElementType = RECORD
           (* the type you want to store *)
        END;
        PSomeElementType = POINTER TO SomeElementType;
        (* now you have a type compatible to 'Item' *)
   (* now you must provide the operations, which are then associated with your type: *)
   PROCEDURE Assign( e: PSomeElementType): PSomeElementType;
      BEGIN (* ... *) END Assign;
   PROCEDURE Compare( e1, e2: PSomeElementType): BOOLEAN;
      BEGIN (* ... *) END Compare
   PROCEDURE Dispose( VAR e: PSomeElementType);
      BEGIN (* ... *) END Dispose;
   VAR S: Set;
       e: PSomeElementType;
       T: TypeID;
BEGIN
   (* ... *)
   T := TCreate( "aType", AssignProc( Assign), CompareProc( Compare), DisposeProc( Dispose));
   S := Create( T);
   Insert( S, e);
   (* ... *)
END ClientModule1.
```



2.2. Solutions with changes to the language

The solution by /Zimm 85/ only slightly changes the definition of Modula−2 in proposing one−way compatibility between procedure types. Since this approach leads to similar results as the TMM method described above we will not consider it any further.

In /Reyn 87/ an INCLUDE statement is introduced, which may be placed at any position

where a local module is allowed. This INCLUDE statement is to be in−line expanded at compiletime by a file containing the declaration of a local module. This local module acts as a generic module. Therefore, it imports the generic, actual parameters from its enclosing module to support generic operations using these parameters.

This method ensures strong typing, but produces a lot of redundant code. This leads to the same maintenance problems as reported for the EM method. Since there are no explicit interfaces to the generic modules this method leads to an obscure programming style. Especially the deklaration of formal generic procedure parameters (Equal in example 2.4) is very clumsy.

Example 2.4: The method described by /Reyn 87/

```
MODULE GenericSet;     (* stored on file Set.TYPE *)
   IMPORT  (* what follows are the formal generic parameters: *)
      SomeType, Equal (* ... *);
   EXPORT QUALIFIED   (* what follows are the capabilities of the generic set: *)
      Set, Create, Insert (* ... *);
   TYPE     (* the signature of the formal generic procedure parameter 'Equal': *)
      EqualProc = PROCEDURE( SomeType, SomeType): BOOLEAN;
   (* what now follows is the implementation of the generic set: *)
   TYPE Set = (* ... *)
   PROCEDURE Create(): Set;
      BEGIN (* ... *) END Create;
   PROCEDURE Insert( VAR S: Set; data: SomeType; Equal: EqualProc);
      BEGIN (* ... *) END Insert;
   (* ... *)
END GenericSet.
MODULE ClientModule;
   (* ... *)
   TYPE     (* what follows are the actual generic parameters: *)
      SomeType = REAL;
   PROCEDURE Equal( r1, r2: REAL): BOOLEAN;
      BEGIN RETURN( r1 = r2) END Equal;
   INCLUDE Set.TYPE;   (* the file 'GenericSet' is defined in *)
   VAR      (* now sets of 'REALS' can be used: *)
      aRSet: GenericSet.Set;
      r: REAL;
   BEGIN
      (* ... *)
      aRSet := GenericSet.Create();
      GenericSet.Insert( aRSet, r, Equal);
   END ClientModule.
```

Some major language extensions are proposed by /LeSp 87/ and /Craw 87/. Both suggest to enrich Modula−2 with constructs that allow the declaration of generic types and their instantiation similar to the concepts found in Ada®. Since the extensions proposed by /LeSp 87/ are a subset of those proposed by /Craw 87/ we will only discuss the latter approach.

2.2.1. The Approach of /Craw 87/

/Craw 87/ permits the declaration of ASSOCIATED and GENERIC types as well as GENERIC procedures. Generic procedures are procedures with associated and/or generic types as parameters. Each generic type has associated a list of associated types with it acting as formal parameters to its instantiation. A preprocessor uses this additional information to perform type checking on associated and generic types, resp. the instances of generic types. Thereafter the preprocessor generates "standard" Modula−2[3] as input to a conventional Modula−2 compiler, treating the parameters of associated and generic types as being of type ARRAY OF BYTE/WORD.

We discuss this proposition in more detail because of its influence on our approach.

Example 2.4: The method described by /Craw 87/ (cf. /Craw 87/)

3. *With "standard" Modula−2 we always refer to /Wirth 85/.*

```
DEFINITION MODULE GenericStack;
   ASSOCIATED TYPE Item;
   GENERIC TYPE Stack( Item);
   GENERIC PROCEDURE Init( VAR S: Stack);
   GENERIC PROCEDURE Pop( VAR S: Stack; VAR E: Item);
   GENERIC PROCEDURE Push( VAR S: Stack; E: Item);
END GenericStack.
```

```
IMPLEMENTATION MODULE GenericStack;
   FROM Storage IMPORT TSIZE, ALLOCATE, DEALLOCATE;
   GENERIC TYPE Stack( Item) = POINTER TO StackRecord;
   GENERIC TYPE StackRecord( Item) = RECORD
                  Element: POINTER TO Item;
                  Next: Stack
               END;
   GENERIC PROCEDURE Init( VAR S: Stack);
   BEGIN
      S := NIL
   END Init;
   GENERIC PROCEDURE Pop( VAR S: Stack; VAR E: Item);
      VAR Temp: Stack;
   BEGIN
      IF S <> NIL THEN
         E := S^.Element^; (*1*)
         Temp := S;
         S := S^.Next;
         DEALLOCATE( Temp^.Element, SIZE( E));
         DEALLOCATE( Temp, TSIZE( StackRecord));
      END
   END Pop;
   GENERIC PROCEDURE Push( VAR S: Stack; E: Item);
      VAR Temp: Stack;
   BEGIN
      ALLOCATE( Temp, TSIZE( StackRecord));
      ALLOCATE( Temp^.Element, SIZE( E));
      Temp^.Element^ := E; (*1*)
      Temp^.Next := S;
      S := Temp
   END Push;
END GenericStack.
```

```
MODULE StackUse;
   FROM GenericStack IMPORT
      Stack, Init, Pop, Push;
   TYPE Something = RECORD
           a, b, c: CARDINAL
        END;
        SomeStack = NEW Stack( Something);
   VAR S: SomeStack;
       E1, E2: Something;
BEGIN
   (* ... *)
   Init( S);
   Push( S, E1);
   Pop( S, E2);
   (* ... *)
END StackUse.
```

To obtain correct Modula−2 code the preprocessor cannot substitute all occurrences of `Item` in the same way. The type `ARRAY OF BYTE/WORD` is only allowed as formal parameter in procedure declarations. Therefore, something like `ADDRESS` has to be used for variable declarations (`ARRAY [0..MAX] OF BYTE/WORD` is not very efficient in terms of space). This results in serious type compatibility problems (*1*) in assigning parameters (of type `ARRAY OF BYTE/WORD`) to local variables (of type `ADDRESS`) or vice versa. These problems can be solved in two ways:

(a) Performing a bytewise copy, which is not very time efficient and tricky to implement (because ”standard” Modula−2 allows the copying of whole words only).

(b) Substituting `ADDRESS` for all occurrences of `Item` (i.e. for the formal procedure parameters too), which restricts the use of the generic ADTs.

Further problems arise when dealing with generic array and procedure types.

Example 2.5: A more complicated example (cf. /Craw 87/)

```
DEFINITION MODULE GenericSort;
   ASSOCIATED TYPE Item;
   GENERIC TYPE SortArray( Item) = ARRAY OF Item;
   GENERIC TYPE CompareItems( Item) =
                  PROCEDURE ( X, Y: Item): BOOLEAN;
   GENERIC PROCEDURE Sort( VAR A: SortArray; LessThan: CompareItems)
END GenericSort.
IMPLEMENTATION MODULE GenericSort;
   GENERIC PROCEDURE Swap( VAR a, b: Item);
      (* exchanges two values of arbitrary type *)
      VAR Temp: Item;
   BEGIN
      Temp := a; (*1*)
      a := b;
      b := Temp (*1*)
   END Swap;
   GENERIC PROCEDURE Sort( VAR A: SortArray; LessThan: CompareItems);
      VAR i, j: CARDINAL;
   BEGIN
      FOR i := 0 TO HIGH( A)-1 DO (*2*)
         FOR j := HIGH( A)-1 TO i BY -1 DO (*2*)
            IF LessThan( A[j-1], A[j]) THEN (*3*)
               Swap( A[j-1], A[j]) (*3*)
            END
         END
      END
   END Sort
END GenericSort.
MODULE SortUse;
   FROM GenericSort IMPORT SortArray, Sort;
   TYPE Something = (* ... *);
   SomeArray = NEW SortArray( [0..100], Something);
   PROCEDURE SomethingLessThan( a, b: Something): BOOLEAN;
      BEGIN (* ... *) END SomethingLessThan;
BEGIN
   (* ... *)
   Sort( SomeArray, SomethingLessThan); (*4*)
   (* ... *)
END SortUse.
```

Generating correct Modula−2 code for the procedure `Swap` in this implementation causes the same problem as mentioned in the previous example (*1*). In addition there occur problems in generating correct Modula−2 code for the `FOR` loop, because `HIGH( A)` is the number of bytes in `SortArray`, when substituting `ARRAY OF BYTE/WORD` for `SortArray` (*2*). Furthermore, the actual parameters in the procedure calls `LessThan` and `Swap` are not the expected ones (*3*). These problems can only be solved in rewriting the sorting operation and supplying it with an additional parameter (this parameter may be added by the preprocessor) holding the current size of the type `Item`. Next, the size information must be transferred to the swap operation to allow the creation of a correct sized local variable.

Another problem stems from parameters of procedure type. Although it should cause no problems to substitute `ARRAY OF BYTE/WORD` for the formal item type(s) compilers run into trouble because of the additional `HIGH( ...)` information for open array parameters on the parameter stack. Some compilers even raise an error if formal procedure types with `ARRAY OF BYTE/WORD` parameters are supplied with actual procedures with parameters of concrete types ((*4*), but this is due to uncorrect implementations of Modula−2). This problem could be solved by introducing one−way compatibility between procedure types as proposed by /Zimm 85/. Examples for generic sorting routines in correct Modula−2 code are given in /Wien 84/, which address both problems stated above. First, parameters are added to the sorting routines to support work space for the (generic) objects to sort. Second, actual and formal

parameters of type `ARRAY OF WORD` are used for the procedures that compare (generic) objects.

From the examples (2.4, 2.5) given below we learnt that this form of genericity is useful only for simple cases of genericity. Example 2.4 showed that even the simplest form of genericity causes problems. If the actual generic type (`Something` in Example 2.5) must fulfil some requirements given in the form of generic procedure types (`CompareItems` in Example 2.5), the preprocessor must be able to perform additional analysis to generate correct code. This analysis can cause problems, if the actual generic type is itself imported from another module. To circumvent these problems object oriented extensions ca be used to bind type information (i.e. the class an object belongs to) to variables (objects).

3. OBJECT−ORIENTED EXTENSIONS TO MODULA−2

It is already possible to use an object oriented programming style in conventional, procedural programming languages (like Modula−2), if these languages provide data encapsulation, dynamic data structures and procedure variables. But, e.g., simulating inheritance, which is usually done using record type extension mechanisms, is extremely erroneous and demands utmost programming discipline. To introduce object orientation to Modula−2 to get more reusable, reliable and extendible software, the programmer must be supported with special tools to achieve type safety and comfortable handling of the object oriented constructs.

In this chapter we will first introduce the approaches proposed by /BeGr 88/ and /Bla 89/, which show the possibility to write object oriented programs in Modula−2 without extending the language. But there are some deficiencies that are not solved by these approaches leading to serious reusability problems. Therefore, we propose some extensions to the syntax of Modula−2. A preprocessor can then solve many of the problems mentioned in 3.1.1, using these extensions. Thus, we hope to achieve an easy, type safe and systematic way of using object oriented constructs in Modula−2.

3.1. Solutions without changing the language

3.1.1. The Approaches of /BeGr 88/ and /Bla 89/

The main mechanisms used to introduce an object oriented programming style to Modula−2 are described in /BeGr 88/ . Their approach is very similar to the one taken in Modula−3 (/Ca.et.al. 88/). The main idea is to define records which contain fields for procedures. According to the object oriented paradigm records of this type may be considered as objects and the contained procedures as their methods[4].

Example 3.1: A `Stack` for complex numbers based on an array

```
DEFINITION MODULE Stack;
   FROM SYSTEM IMPORT ADDRESS;
   FROM GlobalTypes IMPORT ComplexNumber;

   CONST MaxStackSize = 10;

   TYPE
      Object = POINTER TO ObjectDesc;
      PushProcType        = PROCEDURE( ADDRESS, ComplexNumber );
      PopProcType         = PROCEDURE( ADDRESS ) : ComplexNumber;
```

4. We will use the terminology introduced by /GoRo 83/ and /Bla 89/.

```
      ObjectDesc = RECORD
                      Push         : PushProcType;
                      Pop          : PopProcType;
                      Memory       : ARRAY [ 1 .. MaxStackSize ] OF ComplexNumber;
                      Top          : [ 1 .. MaxStackSize ];
                   END (* record *);
END Stack.
(* ---------------------------------------- *)
MODULE Client;
IMPORT Stack;
FROM GlobalTypes        IMPORT ComplexNumber;
FROM ComplexNumbers     IMPORT ComplexAdd;

VAR  SomeStack          : Stack.Object;
     Val1, Val2, Sum    : ComplexNumber;

BEGIN
   (* ... *)
   Val1 := SomeStack^.Pop( SomeStack );
   Val2 := SomeStack^.Pop( SomeStack );
   Sum  := ComplexAdd( Val1, Val2 );
   SomeStack^.Push( SomeStack, Sum );
   (* ... *)
END Client.
```

Notice that an object does not only consist of the data stored in it. The object also contains knowledge about **all** methods which may be performed on this data. Calling a method `M` for an object `O` is done by referencing the corresponding record field `O^.M` and then passing the object itself as first parameter to the method `O^.M( O, ... )` because the method may need access to the object.

Using this mechanism dynamic binding of method names to method implementations is already achieved. One may initialize different 'stack−objects' with different procedures, e.g. some stack objects may refer to a push method which additionally maintains some statistical data. This leads to syntactically equivalent (polymorphic) 'method calls', which indeed execute different procedures.

The second basic idea (to introduce object oriented constructs into Modula−2) is to use (record) type extensions for the simulation of inheritance.

Example 3.2: A '`Stack`'−subclass with additional features

```
DEFINITION MODULE AddingStack; (* IS_A Stack *)
   (* Class AddingStack inherits all features of class Stack *)
   FROM SYSTEM IMPORT ADDRESS;
   FROM GlobalTypes IMPORT ComplexNumber;
   FROM Stack IMPORT MaxStackSize;

   TYPE
      Object = POINTER TO ObjectDesc;

      (* Inherited method types *)
      PushProcType        = PROCEDURE( ADDRESS, ComplexNumber );
      PopProcType         = PROCEDURE( ADDRESS ) : ComplexNumber;

      (* new defined method types *)
      AddProcType         = PROCEDURE( ADDRESS );
      NegateProcType      = PROCEDURE( ADDRESS );
      ObjectDesc = RECORD
                      (* Inherited features *)
                      Memory       : ARRAY [ 1 .. MaxStackSize ] OF ComplexNumber;
                      Top          : [ 1 .. MaxStackSize ];
                      Push         : PushProcType;
                      Pop          : PopProcType;

                      (* new defined methods *)
                      Add          : AddProcType;
                      Negate       : NegateProcType;
                   END (* record *);
END AddingStack.
(* ---------------------------------------- *)
MODULE Client;
   IMPORT Stack;
   IMPORT AddingStack;
```

```
FROM GlobalTypes         IMPORT ComplexNumber;
FROM ComplexNumbers      IMPORT ComplexAdd;

VAR  SomeStack                 : Stack.Object;
     SomeAddingStack           : AddingStack.Object;
     Sum, Val                  : ComplexNumber;

BEGIN
   (* ... *)
   SomeStack := Stack.Object( SomeAddingStack ); (*1*)
   SomeStack^.Push( SomeStack, Val );
   SomeStack^.Push( SomeStack, Val );
   SomeAddingStack := AddingStack.Object( SomeStack ); (*2*)
   SomeAddingStack^.Add( SomeAddingStack );
   (* ... *)
END Client.
```

The used modules containing the whole type definitions (and method implementations) for objects of one type may be seen as `classes`. According to object oriented conventions the type `Stack` is called the 'superclass' of the 'sub−' or 'descendant−' class `AddingStack`. As we see in the example above an object of the subclass `AddingStack` provides all 'features' that are provided by a `Stack` object. We say the class `AddingStack` inherits the features of the class `Stack`. Additionally objects of the class `AddingStack` offer new methods like `Add` and `Negate`, allowing some simple arithmetic operations on stack objects.

An important object oriented concept is that every time an object of a class `C` is expected a subclass object can be used instead. As the class `AddingStack` inherits all features of the class `Stack`, it is possible to use an `AddingStack` object everywhere an object with `Stack` properties is demanded. With respect to the strong type checking facilities of Modula−2 we use in our example (unsafe) type coercion `(*1*)` to assign the `AddingStack` object `SomeAddingStack` to the variable `SomeStack` of type `Stack.Object`[5]. Notice that `SomeStack` still refers to an `AddingStack` object, thus it is allowed to assign it back to `SomeAddingStack`[6] `(*2*)`. This is forbidden in general, because `SomeStack` could refer to a simple `Stack` object not containing the additional fields for the adding features.

We do not discuss the approach of /BeGr 88/ in further detail now, because /Bla 89/ describes a more sophisticated object oriented mechanism. This approach has several advantages, especially from the viewpoint of software engineering, reusability, type safeness, and information hiding.

In /Bla 89/, there are two main differences/advantages to /BeGr 88/. First, /Bla 89/ proposes to use an own (definition) module for every class. This 'class' module contains all type declarations concerning objects of this class. In particular, all methods understood by objects of this class are declared there. Second, instead of storing the methods in every object itself /Bla 89/ proposes to use one special variable (`TheStack`), that contains the information belonging to all objects.

Example 3.3:

```
DEFINITION MODULE Stack; (* IS_A ANY *)
   (* Deferred class describing  an interface for  Stacks of  complex *)
   (* numbers.  No implementation is provided.                        *)
   FROM SYSTEM IMPORT ADDRESS;
   FROM GlobalTypes IMPORT ComplexNumber;
   IMPORT ANY;
```

5. Our manual record type extension mechanism used to simulate inheritance is based on type coercion. This works only if the common prefix of different record types are arranged identically in the storage. (Most compilers do so. Errors may occur if optimisation is switched on.

6. The assignment is not really necessary, because `SomeAddingStack` *still refers to that object although its internal state has changed through the* `Push` *method.*

```
CONST MaxStackSize = 10;

TYPE
   ClassName              = ARRAY [ 1 .. 20 ] OF CHAR;
   Object                 = POINTER TO ObjectDesc;
   ObjectData;            (* opaque type used here *)
   ObjectDesc             = RECORD
                               Do   : InstanceMethods;
                               Data : ObjectData;
                            END (* record *);

   InstanceMethods        = POINTER TO InstanceMethodsDesc;

   (* Inherited instance method types *)
   CopyToProcType         = PROCEDURE( Object, Object );
   CloneProcType          = PROCEDURE( Object ) : Object;
   DisposeProcType        = PROCEDURE( Object );

   (* New defined instance method types *)
   PushProcType           = PROCEDURE( Object, ComplexNumber);
   PopProcType            = PROCEDURE( Object) : ComplexNumber;

   InstanceMethodsDesc    = RECORD
                               (* Pointer to the superclass    *)
                               Super    : ANY.InstanceMethods;
                               Name     : ClassName;

                               (* Inherited instance methods   *)
                               CopyTo   : CopyToProcType;
                               Clone    : CloneProcType;
                               Dispose  : DisposeProcType;

                               (* New defined instance methods *)
                               (* DEFERRED *)  Push     : PushProcType;
                               (* DEFERRED *)  Pop      : PopProcType;
                            END (* record *);

   (* Standard procedures performing runtime type checks *)
   PROCEDURE Convert( Obj : ADDRESS ) : Object;
   PROCEDURE Member ( Obj : ADDRESS ) : BOOLEAN;

   VAR
      TheClass       : InstanceMethods;
END Stack.
```

Using a separate module for every class has many advantages not only from the viewpoint of software engineering. It facilitates the use of a special, class global variable `TheClass`, which stores common information for all objects of this class, and it allows to assure proper initialization of this common data at program start using the module initialization part.

/Bla 89/ assumes that all objects of one class should behave similar, i.e. should use the same method procedures. Thus the references to the methods are always the same for all objects of one class. Consequently, /Bla 89/ proposes to store the methods provided by a class in its `TheClass` variable. Thereby, it is no longer necessary that every object itself contains a big, storage wasting record part for its methods. Instead of this only the pointer `Do` is necessary which references the corresponding `TheClass` variable. Hence not only much storage space is saved, but also a better correspondence to usual object oriented languages is achieved. These advantages are achieved by the cheap price of an additional dereferencing operation for a method call. A method call is now done by `O^.Do^.M( O, ...)` (which explains the choice of the name for the method pointer).

In /BeGr 88/ an object consists of one record containing both, the method and the data fields. This has the disadvantage that these data fields, which contain internal information, are directly accessible for every client module. This is not acceptable from the viewpoint of software engineering. To avoid this problem /Bla 89/ proposes the use of an Modula−2 opaque data type for the instance variables. If in some cases a Client module needs access to an object's data a method must be used for that purpose. Thus in /Bla 89/ an object consists only of the `Do` pointer to its `TheClass` record and of an entry for its opaque data. This intro-

duces information hiding and achieves better reliability compared to /BeGr 88/[7].

Now we return to the common information stored in the variable `TheClass`. Perhaps the most important field of it is the pointer `Super`. This pointer links the current variable to the `TheClass` variable of the corresponding superclass. Since every class contains a pointer to its superclass (eventually to the pseudo class `Any`), a hierarchy is built which is very similar to the usual class hierarchies build by object oriented languages. In our case this hierarchy is a tree, because multiple inheritance is forbidden.

This hierarchy is used for many purposes. Via the `Do` pointer the objects refer to their corresponding `TheClass` record within the class hierarchy. Thus every object has access to the `Super` pointer of its class and to its `supermethods`. This may be used if one wants a descendant class's method `M` to behave very similar to the inherited method `M` but to do some additional work. In such a case, the method `M` first should call its supermethod by `O^.Do^.Super^.M( o, ...)`. This performs the inherited method and the old behavior is reached. After that call the source code for the additional work may be written, and that's all to do. Especially, one need not to duplicate the whole inherited method source what would be unpleasant from the viewpoint of software engineering.

Furthermore, using the hierarchy built by the `Super` pointers it is possible to provide runtime type checking facilities. Runtime type checking is necessary because a variable of the class `Stack.Object` may refer to an object of the class `AddingStack.Object` (or of a subclass of that class). Assigning a `Stack.Object` variable to an `AddingStack.Object` variable is possible if and only if the `Stack.Object` variable actually refers to an `AddingStack.Object`. To assure this we use the `Convert` routine which works as follows:

A class `SC` is superclass of a class `C` if and only if `SC` can be reached by following the `Superclass` pointer repeatedly. Otherwise the class `Any` is reached.

If one tries to convert an object of a class `C` into an object of a class `C2` and the class `C` is not a subclass of `C2`, then the `Convert` routine will perform some error handling and abort execution.

In addition to this `Convert` routine, every class provides a `Member`−ship test routine, which also uses the class hierarchy.

Now we will discuss the inheritance mechanism of /Bla 89/. Like /BeGr 88/ inheritance is simulated by a record type extension mechanism.

Example 3.4:

```
DEFINITION MODULE AddingStack; (* IS_A Stack *)
   (* This class  implements a Stack  for Complexnumbers, satisfying *)
   (* the declarations of the deferred  class Stack.  The superclass *)
   (* Stack  then  is  extended  by  procedures   allowing    simple *)
   (* arithmetic operations on such stacks.                          *)
   FROM SYSTEM IMPORT ADDRESS;
   FROM GlobalTypes IMPORT ComplexNumber;
   IMPORT Stack;
   CONST MaxStackSize = 10;
   TYPE
      ClassName              = ARRAY [ 1 .. 20 ] OF CHAR;
      Object                 = POINTER TO ObjectDesc;
      ObjectData;
      InstanceMethods        = POINTER TO InstanceMethodsDesc;
      (* Inherited method types *)
      CopyToProcType         = PROCEDURE( Object, Object );
      CloneProcType          = PROCEDURE( Object ) : Object;
      DisposeProcType        = PROCEDURE( Object );
```

7. *Note, in the approach of /BeGr 88/ there's no information hiding for the* `TheClass` *variable.*

```
      (* Inherited and redefined instance methods *)
      PushProcType           = PROCEDURE( Object, ComplexNumber );
      PopProcType            = PROCEDURE( Object ) : ComplexNumber;

      (* New defined instance method types *)
      AddProcType            = PROCEDURE( Object );
      NegateProcType         = PROCEDURE( Object );
      InstanceMethodsDesc    = RECORD
                                  Super    : Stack.InstanceMethods;

                                  (* class name used for messages *)
                                  Name     : ClassName;

                                  (* Inherited methods *)
                                  CopyTo   : CopyToProcType;
                                  Clone    : CloneProcType;
                                  Dispose  : DisposeProcType;

                                  (* Inherited and redefined methods *)
                                  Push     : PushProcType;
                                  Pop      : PopProcType;

                                  (* New defined methods *)
                                  Add      : AddProcType;
                                  Negate   : NegateProcType;
                               END (* record *);

      ObjectDesc             = RECORD
                                  Do   : InstanceMethods;
                                  Data : ObjectData;
                               END (* record *);

   (* Standard procedures *)
   PROCEDURE Create( ActStackSize : CARDINAL) : Object;
   PROCEDURE CONSTRUCTOR( Self : Object; ActStackSize : CARDINAL);

   VAR
      TheClass       : InstanceMethods;
END AddingStack.
```

Notice that the first fields of the record type `AddingStack.InstanceMethodsDesc` exactly match the type `Stack.InstanceMethodsDesc`. This correspondence is extremely important. With the help of a `Convert` routine one may assign an `AddingStack` object to a variable of type `Stack.Object`, which is a legal operation. But the Modula−2 compiler assumes that a `Stack` object is referenced by this variable. This works correctly if and only if the `Stack.Object` structure (and the included `Do` and `Data` structures) exactly matches the beginning of an `AddingStack` object (and it's `Do` and `Data` structures). The reader may imagine what happens if one extends the type `Stack.InstanceMethodsDesc` by a method or a data field without paying attention to the subclass `AddingStack`. An `AddingStack` object must also inherit the data fields of a `Stack` object, because it uses inherited methods, which access these data fields for objects they work on. Thus, the simulation of inheritance in Modula−2 is only possible if the extension of the concerned record types is done correctly.

Defining the described data types and maintaining the type extension for inheritance is the main work in writing the definition module of a class. For the implementation module the following has to be done. First one must inherit the object data type from the superclass, i.e., one must copy (and eventually extend) the corresponding type definition, cf. appendix A. Second, one has to think about the implementation of the methods. Fortunately, the subclasses (`AddingStack`) inherit not only type definitions from their superclasses, but also the method implementations are inherited. How this works will be shown in further details in the discussion about the initialization of the `TheClass` variable. Hence in general only new methods have to be implemented in a subclass. But there are two exceptions, where one has to provide a new implementation for an inherited method:

(1) if the method of the subclass has to behave differently from the inherited method. In this case one may redefine the inherited method. I.e., one provides a new implementation for this routine and assigns it to the corresponding method variable. But even now

code duplication is avoided. The new implementation may use the inherited method using the `Super` pointer as described above.

(2) if the method is deferred, i.e., the superclass has no implementation of this method itself but requires the subclasses to have one. In our example the class `Stack` has deferred features. It does not provide a implementation for stacks itself but only defines properties common for all stacks. Different descendants of this class may implement stacks in different ways, but they all will at least provide the routines `Push` and `Pop` with the given signature. Since the father class does not implement the method, naturally the descendants have to do this.

In general we can say that in /Bla 89/ a subclass inherits not only the interface of its superclass but also the implementation of the object methods is inherited. Hence in most cases code duplication is avoided and program maintenance is facilitated, and good analogy to common object oriented properties is reached. But in /Bla 89/ this sophisticated mechanism only works for the object's methods. For the remaining standard routines such as `Convert`, `Member` and `Create`, which are also in some way 'inherited' from the superclass, the complete source code is simply copied into the subclass. Thus code is duplicated, storage space is wasted, maintenance becomes difficult. Additionally, the module where the source code is copied in has to provide the imports, global variables, called procedures and type definitions which are used in the inserted sources. Serious errors may occur if the expected code environment differs for a subclass.

The next aspect of implementation we have to discuss is the `Create` routine. While the global variable `TheClass` exists only once and hence can be initialized at program start, there may be several objects (instances) of a class. Since /Bla 89/ uses pointer semantics, these objects have to be allocated and initialized explicitly by the `Create` routine. To initialize the object's data correctly the `Create` routine needs appropriate parameters. But this parameter signature will probably change because subclasses may define new data fields.

While the `Convert` and `Member` routine just may be copied to a subclass and no code modification is necessary, for the `Create` routine the signature and the routine body has to be modified. The body of the `Create` routine of a subclass must contain initialization statements for the whole data of the subclass object (i.e. for the inherited data and for the new defined). Hence for the `Create` routine not only the problem of code duplication arises but also the maintenance problem we already discussed for the type extension mechanism simulating inheritance.

After these unsolved problems we will now discuss an outstanding advantage of /Bla 89/. One of the most appealing benefits of the use of an own module for every class is, that one can use the module initialization parts of the classes to build up the class hierarchy and to initialize the used data structures. Hence a client need not worry about the correct values of the used procedure variables, of the `TheClass` variable, and the `Super` pointer. No `Init` routine must be called or could be forgotten. But the programmer of the class himself is responsible for this work and has to decide, which actual routine he wants to assign to the method variables. At default he may use the inherited method implementations for this purpose:

```
WITH TheClass^ DO
   ...
      WITH InstanceDo^ DO
         Super := SuperClass.TheClass^.InstanceDo;
         ...
         M := MProcType( Super^.M ); (* 1 *)
         ...
```

```
            M2 := OwnM2Proc; (* 2 *)
            ...
        END; (* WITH InstanceDo *)
    ...
END; (* WITH TheClass^ *)
```

If an inherited method is used (1) it can be reached again using the `Super` pointer. The type casting is necessary because of the object parameter, which changes to the current subclass object type. For the initialization of redefined or deferred methods (* 2 *) own routines are used. For the full initialization part cf. appendix A.

Summarizing the above discussion the approach of /Bla 89/ may be characterized as follows:

- separate modules for every class
- an explicit class hierarchy
- runtime checked type coercion
- one common method suite referred by all objects via the `Do` pointer
- use of opaque data types for the object's data, thus introducing information hiding
- sophisticated inheritance mechanisms avoiding code duplication, allowing redefined and deferred methods

The approach of /Bla 89/ has several advantages and is closer to standard object oriented concepts than the one of /BeGr 88/. And it solves many problems raised by the introduction of an object oriented programming style in Modula−2 in an elegant manner. But some serious problems still remain:

- because Modula−2 does not provide (record) type extension facilities, inheritance has to be managed manually. Introducing a new class means copying an old one, changing the names consistently, and then extending and modifying the inherited method records and / or instance variables records in a correct manner. If one extends / modifies an existing class one oneself has the responsibility to propagate these changes to all descendant classes to avoid serious system crashes. This has to be done manually.
- a method call is easy to read but awkward to write and it is a source of errors because of the repetition of the object's name. The call

  ```
  x^.Do^.m( y )
  ```

  may have unpredictable effects. E.g., `x` could actually refer to a subclass object and `y` to a superclass object. Hence a subclass routine is called which can be redefined and may access subclass variables which are not provided by the superclass object `y` which is passed as a parameter.
- /Bla 89/ did not mention that for the instance variables similar inheritance mechanisms are necessary.
- /Bla 89/ does not introduce a good solution for the inheritance of class routines such as `Convert` and `Member`.

In the following section we try to find solutions for some of these problems.

3.1.2. Extensions to /Bla 89/

Most of the topics of the summary above will be discussed in section 4. In this chapter we will make a proposal for the last of these topics. The problem is that the source code of the `Convert` and `Member` routine must be copied for every class.

If one wants to avoid this code duplication one has to provide some class global procedure

variables to store a pointer to these routines. Additionally there may be the case, that some of the object data items must be shared between all objects of a class. Such common information should be stored only once in a data area accessible by all objects. According to the concepts of information hiding this data should be opaque and routines for accessing it have to be provided.

Descendant classes then also inherit these global properties and may redefine it. Thus, at least for the inherited access routines the same inheritance mechanisms must be supported as for the object methods of a class[8].

In many object oriented languages this situation is well-known. These languages consider such class global data and routines as usual objects, called class objects, with all the usual mechanisms provided for any object. In Smalltalk these class objects are not distinguished from normal objects, called instance objects. Hence we will get a more sound concept if we handle the class data and the corresponding routines as features of a class object. Then `Convert` and `Member` become class methods, which solves the code duplication problem for these routines.

To achieve these advantages we have to provide the same data structures and mechanisms we used for instance objects for class objects too. This means the `TheClass` variable, which represents the class object, has to contain a `Data` pointer to the opaque class data and a `Do` pointer referring to the class methods. A call to a class method is done like a call to an instance method by (cf. appendix A) `TheClass^.Do^.ClassMethod( TheClass, ...)`. A `Super` pointer is provided and inheritance has to be maintained using type extension. In our concept the distinction between a class object and an instance object is, that class objects cannot be created dynamically. There will be only one object for every class which is created at program start. Therefore, no `Create`, `Convert` or `Member` methods are provided for class objects. Since an object should have access to its class (it may contain common data) the instance objects get a new pointer (`TheClass`) to their class object. An object thus may call a method of its class by `Object^.TheClass^.Do^.ClassMethod( Object^.TheClass, ...)`.

To sum up, the extensions we proposed in this section will lead to a more sound object oriented programming concept for Modula−2 and increase the expressive power of this approach. This is reached by doubling the data structures we use. The now used data structures are shown in appendix A. Indeed the used mechanisms are so complex and especially the manual type extension is so erroneous, that the original problem, to get more reusable and easier maintainable software is not reached until now. We hope this problem will be solved by the proposals of the next section.

3.2. Solutions with changes to the language

The approach of /Bla 89/ and our extension to it suffer from a number of serious practical problems already mentioned in the summary of chapter 3.1.1. These problems are the complexity of the type extensions and therefore the data structures which have to be maintained manually, and the complicated method access mechanisms. We now propose some ideas to solve several of these problems. The main idea is to add some new syntax to Modula−2 to allow a simpler and saver use of the object oriented constructs. Then we will use a preproces-

8. For the data there may be both cases: that one wants to have own global variables for the descendant class or that one wants the descendant class to share the inherited variables with its superclass. Our concept follows the first case, but the second is also possible.

sor to analyze this new syntax and produce conventional Modula−2 source code in a style similar to chapter 3.1.

3.2.1. Our syntax for definition of classes

We think the designer of a new class should not be forced to repeat all the definitions of the corresponding superclass. Instead he should only denote this superclass and our preprocessor will then produce code for the inherited features. If one combines this with a utility like the UNIX®−make, extensions or modifications of a superclass are propagated automatically to all subclasses and thus the type extension problem is solved.

We also want to free the programmer from work which has to be done repeatedly and/or at several places. E.g. in /Bla 89/ for the introduction of a new method one has to

- define a type describing its signature
- define a field in the method suite
- implement the routine
- initialize the method field with this routine

In our approach one only defines the new method together with its signature and provides the implementation. The preprocessor does the remaining work. This especially means that the preprocessor takes care of the proper initialization of the method suits, hence a priori avoiding possible user mistakes.

Additionally we free the programmer from writing all the standard definitions for the used data structures, which are equal for every class. Except for debugging purposes one does not need to know these complex structures in full detail, because our preprocessor manages them.

Last but not least we will explicitly denote deferred and redefined methods and introduce the above discussed conceptual extensions by using distinct parts for class and instance features.

Example 3.5:

```
CLASS ANY;
   FROM System IMPORT ADDRESS;
   CLASS_FEATURES
      METHODS
         Convert( ADDRESS ) : Object;
         Member( ADDRESS ) : BOOLEAN;
      END METHODS;
   END CLASS_FEATURES;

   INSTANCE_FEATURES
      METHODS
         CopyTo( Object );
         Clone() : Object;
         Dispose();
     END METHODS;
   END INSTANCE_FEATURES;

   CONSTRUCTOR( );

IMPLEMENTATION_PART (* ------------------------------------------*)
           (* ... *)
END ANY;

(* ---------------------------------------------------------------*)
CLASS Stack IS_A ANY;
   FROM GlobalTypes IMPORT ComplexType;
   CLASS_FEATURES
      METHODS
         GetNoOfInstances( ) : CARDINAL;
      END METHODS;
      VARIABLES
         NoOfInstances : CARDINAL := 0;
```

```
         (* This class has no direct instances, but the indirect ones are counted *)
      END VARIABLES;
   END CLASS_FEATURES;

   INSTANCE_FEATURES
      METHODS
         DEFERRED Push( ComplexType );
         DEFERRED Pop( ) : ComplexType;
      END METHODS;
   END INSTANCE_FEATURES;

   CONSTRUCTOR( );

IMPLEMENTATION_PART (* ------------------------------------------------*)
   (* ... *)
END Stack;

(* ----------------------------------------------------------------*)
CLASS AddingStack IS_A Stack;
   (* The class AddingStack inherits all stack features          *)
   FROM ComplexNumbers IMPORT ComplexType;
   TYPE MemoryType; (* Opaque Type *)
   INSTANCE_FEATURES
      METHODS
         (* redefine Clone and Dispose to count NoOfInstances *)
         REDEFINE Clone( ) : Object;
         REDEFINE Dispose( );
         REDEFINE Push( ComplexType);
         REDEFINE Pop( ) : ComplexType;

         (* Procedures for stack arithmetic *)
         Plus( );
         Negate( );
      END METHODS;

      VARIABLES
         Memory        : MemoryType;
         Top           : CARDINAL;
         MaxStackSize  : CARDINAL;
      END VARIABLES;
   END INSTANCE_FEATURES;

   CONSTRUCTOR( CARDINAL );

IMPLEMENTATION_PART (* ------------------------------------------------*)
   FROM Storage IMPORT ALLOCATE, DEALLOCATE, TSIZE;
   IMPORT ComplexNumbers;
   FROM GlobalTypes IMPORT MaxCard;

   TYPE
      MemoryType = POINTER TO ARRAY[ 1 .. MaxCard ] OF ComplexNumber;

   PROCEDURE CONSTRUCTOR( Self : Object; ActStackSize : CARDINAL );
   BEGIN
      (* First call the CONSTRUCTOR of the super class *)
      (* in order to count indiredöct instances        *)
      SuperDo( Self, CONSTRUCTOR );

      (* Now do own initialization *)
      WITH Data( Self ) DO
         (* Initialization of new  defined  object  data. This Stack *)
         (* uses a dynamically sized array.                          *)
         ALLOCATE( Memory, SIZE( ComplexNumber ) * ActStackSize );
         Top := 0;
         MaxStackSize := ActStackSize;
      END; (* WITH Data( Self ) *)

      (* Here we increment the class variable NoOfInstances *)
      INC( Data( TheClass, NoOfInstances ) );
   END CONSTRUCTOR;
   (* ... *)

   (* REDEFINED *)
   PROCEDURE Push( Self : Object; SomeNumber : ComplexNumber );
   BEGIN
      WITH Data( Self ) DO
         IF (Top >= MaxStackSize) THEN
            WriteLn; WriteString( "Stack overflow!" );
         ELSE
            INC( Top);
            Memory^[ Top ] := SomeNumber;
         END; (* IF Top >= MaxStackSize *)
      END; (* WITH Data( Self ) *)
   END Push;
```

```
   (* New defined instance methods ---------------------------------- *)
   PROCEDURE Add( Self : Object);
   BEGIN
      WITH Data( Self ) DO
         IF (Top < 2) THEN
            WriteLn; WriteString( "Stack underflow!" );
         ELSE
            Memory^[ Top - 1 ] := ComplexAdd( Memory^[ Top - 1 ], Memory^[ Top ]);
            DEC( Top );
         END; (* IF Top >= 2 *)
      END; (* WITH Data( Self ) *)
   END AddProc;
END AddingStack;
```

The whole description of a class is divided into the class definition part and the class IMPLEMENTATION_PART. These two parts correspond to the definition and the implementation modules of Modula−2.

The class definition part contains the most important informations for the preprocessor. All new extending (and the REDEFINED) features are declared here. The preprocessor checks whether new feature names collide with inherited ones and whether redefined features are of the correct type. The class definition part is divided into the CLASS_FEATURES and the INSTANCE_FEATURES declarations. These feature parts are again divided in the METHODS− and the VARIABLES− part.

In a METHODS part the name of a method and its parameter types are described. Only the standard object parameter is omitted, it will be added by the preprocessor. The keyword REDEFINED precedes an inherited method for which a new implementation is desired. During this redefinition the signature of the method must not be changed (except of the generated object parameter). We are more restrictive in this point as e.g. EIFFEL (/Mey 88/), because of the serious security holes in the static type checking algorithm of that language, which are described in /Cook 89/.

A method preceded by DEFERRED has no implementation in the current class. If a class contains a (probably inherited but not redefined) deferred method it can only be used for inheritance. It is not useful to create instance objects for such a class. Thus the preprocessor does not generate a `Create` routine, and deferred method variables (which could be reached using `Super`) will refer to an error handling procedure which generates a superclass error and terminates execution. The preprocessor checks whether for every new but not deferred and for every redefined method an implementation is provided in the IMPLEMENTATION_PART of that class.

Although the class definition part contains the list of new class and instance variables the preprocessor generates opaque record types for them in order to prevent unauthorized access. They can be used in the class implementation part(s) only. The new variables are contained in the definition part just to facilitate the preprocessor's work of handling the inheritance mechanism for them. Clients can access the class and interface variables only through methods.

The declaration of a class variable includes its initialization. Although a class is seen as an usual object, this class object is created only once at program start. No creation at runtime is possible. Thus only one initialization of the class variables is necessary and this can be determined in the class variables part. Notice that not only constant expressions are allowed there, because these expressions are copied into the initialization part of the corresponding module.

Conversely there may exist several instance objects of one class, which may be initialized differently at runtime. Therefor, every class has to support a special CONSTRUCTOR routine,

which initializes the instance variables, cf. example 3.5. For every class the CONSTRUCTOR may have arbitrary, individual parameters to allow individual initialization of instance objects at runtime. We believe it is necessary to allow arbitrary changes to the signature, because the initialization of a subclass object is in general not only an extension to the initialization of the superclass. Especially the parameter signature may even shrink within a subclass, because some items may become constant or may be derived from some other, new parameters. Since the signature of the CONSTRUCTOR (and of the Create routine) changes it is not contained in one of the method suits of the class ANY as the routines Convert, Clone or Dispose. The preprocessor generates the routine Create for every class (not containing deferred methods). This Create routine first allocates storage for the new instance object and initializes several common variables (Super, Do, ...). Then the Create routine calls the user provided CONSTRUCTOR for individual initialization of the instance variables. If the CONSTRUCTOR needs several actual parameters the preprocessor generates a Create routine with the same parameter signature. Then the actual parameters of the call to the Create routine are used as actual parameters for the call of the CONSTRUCTOR (and only therefore).

Example 3.6:

```
PROCEDURE Create( ActStackSize : CARDINAL) : Object;
   (* Same parameter signature as CONSTRUCTOR *)
   VAR
      TmpNewObj : Object;
BEGIN
   (* Create an AddingStack object with memory of size ActStackSize *)
   NEW( TmpNewObj );
   TmpNewObj^.Do := TheClass^.InstanceDo;
   NEW( TmpNewObj^.Data );
   (* Now call the CONSTRUCTOR for data initialization. *)
   (* Use Create's parameter for this call.             *)
   CONSTRUCTOR( TmpNewObj, ActStackSize);
   RETURN TmpNewObj;
END Create;
```

Remember that the inherited instance variables must be initialized too, but are opaque. Thus every CONSTRUCTOR must call the CONSTRUCTOR of its superclass for initialization of the inherited variables.

```
PROCEDURE CONSTRUCTOR( Self : Object; ActStackSize : CARDINAL);
BEGIN
   (* ... *)
   SuperDo( Self, CONSTRUCTOR( ) ); (* cf. macros in chapter 3.2.2 *)
   (* ... *)
END CONSTRUCTOR;
```

We decided that this call must be managed by the user himself and is not generated by our preprocessor. The only way we found a preprocessor could generate a call to the super–CONSTRUCTOR would have been to allow only extension of the parameter list of this routine. Then in subclasses we could have used the first parameters to pass them to the supercall of CONSTRUCTOR. We believe that this is too restrictive because of the arguments given above. As we were not able to reach full flexibility we refrained from generating such a call. The only help our preprocessor can give here is to test if the body of the user provided CONSTRUCTOR contains at least one call to the CONSTRUCTOR of its superclass.

While the definition module of a class is generated in most parts the implementation module is just a copy of the IMPLEMENTATION_PART of the class definition. Only some modifications and extensions are made. These are:

- the definitions of the opaque types are inserted
- the initializations for the whole data structures are inserted

- the imports are completed
- the completeness of the method implementations is checked
- the used macros are expanded, cf. chapter 3.2.2

In the implementation part the standard object parameter for the methods is not omitted to allow the programmer to use his own names to access this object, however we used the name `Self` everywhere.

We hope that until now our proposal offers acceptable solutions for the problems of

- maintaining inheritance
- avoiding code duplication for `Convert` and `Member`
- defining the complex data structures used
- initialization of these complex data structures

The problem of easy access to the used data structures is discussed in the following chapter.

3.2.2. Some macros to facilitate method calling

Until now we proposed mechanisms to declare and initialize the complex data structures we use to simulate object oriented programming in Modula−2. What we need now is an easy and safe access to these data structures. Especially a call to a super method is not easy to write and read without some help. But as we discussed at the end of chapter 3.1 even the simple method call

```
X^.Do^.M( X, ... )
```

is tedious and erroneous because of the repetition of the object's name. We hope to give a solution for these problems using the following simple macros.

We use `Obj` for instance objects, `Class` for class objects, `ObjOrClass` for both, `Field` for data fields, and `Meth` for method names.

Abbreviation: `Data( ObjOrClass )`

Expansion: `ObjOrClass^.Data^`

Example:
```
WITH Data( ObjOrClass ) DO
   INC( NoOfInstances )
END; (* WITH Data( ObjOrClass ) *)
```

Comment: For completeness only

Abbreviation: `Do( ObjOrClass )`

Expansion: `ObjOrClass^.Do^`

Example:
```
WITH Do( ObjOrClass ) DO
   Push( ObjOrClass, Complex1 );
   Push( ObjOrClass, Complex2 )
END (* WITH Do( ObjOrClass ) *)
```

Comment: Notice the standard object parameter, which has to be written explicitly here. This is inconsistent in some way. This macro may cause errors because the object parameter is repeated. But it is very practical.

Abbreviation: `Do( ObjOrClass, Meth( ... ) )`

Expansion: `ObjOrClass^.Do^.Meth( ObjOrClass, ... )`

Example:
```
Do( ObjOrClass, Push( Complex1 ) );
Do( ObjOrClass, Push( Complex2 ) );
```

Comment: This is the basic macro for calling a method. It works for class and instance objects. Notice that it avoids the repetition of object parameter.

Abbreviation: `SuperDo( Obj, Meth( ... ))`

Expansion: `Obj^.Do^.Super^.Meth(`

```
Do( Obj^.Do^.Super^.TheClass, Convert( Obj ),... );
```

Example: `SuperDo( Obj, DisplayData( ) );`

Comment: In previous examples of the supercall we did not mention, that for a correctly typed supercall one has to `Convert` the object parameter to a superobject. This can be done using the `TheClass` reference contained in the method suit for access to the correct class method `Convert`. Here, we think, the macro mechanism is very helpful. E.g.,one may try to write the above without using the `Do` macro.

Abbreviation: `ClassDo( Obj, ClassMeth( ... ) )`

Expansion: `Obj^.Do^.TheClass^.Do^.Meth( Obj^.Do^.TheClass, ... )`

Example: `ClassDo( Obj, IncNoOfInst( )`

Comment: Since every instance object knows of its class, this macro easily allows to call a class method directly using an object. Notice that the actual class of the referenced object is used, which is not known at compile time.

Using these macros one does not need to know the data structures we use to simulate the object oriented facilities we provide in full detail (again, except for debugging purposes). Thus, together with the generation of the type definitions for these data structures we think it is easy to use our concepts. For those who want to know the data structures in full detail they are explained in Appendix A.

4. CONCLUSION

Due to our experience, a language should include both, genericity and object oriented concepts (cf. /Phil 91/). At all, genericity and and object oriented concepts may be used independently or/and together in the same Modules. There may be object oriented classes which use a generic base type. Or the generic base type of a module may be a class type, and we demand that this generic type have to be instantiated by one of it's descendant types.

As we have shown, it is possible to introduce and make use of genericity and object oriented concepts in Modula−2. While most of our ideas may be utilized just by the means of a new programming style and some discipline, our proposal includes some simple tools supporting the programmer in using our techniques. Since we restricted ourselves to such simple and directly usable techniques, there are still open problems:

- Our tools do not guarantee the correct use of a generic Module.
- Generic procedure parameters are not sufficiently supported.
- Our record extension mechanism used for the simulation of inheritance is simple but not save. It will not work if you use an optimizing compiler, which performs some resorting of record components for better address alignment. Therefore, and
- for the introduction of multiple inheritance some more sophisticated (and more difficult to use) techniques are necessary.

Solutions to these problems will finally lead to a complete new language (Modula−2++). Instead of our simple tools a full compiler would be needed (as for C++) which should use Modula−2 as an intermediate (target) language (like C for C++). Even the techniques used in the implementation of C++ for the introduction of (multiple) inheritance could be used, cf. /ElStr 90/. This approach would have the advantage that the desired generic and object oriented concepts could be used together with a large amount of existing Modula−2 code within the same (existing) project.

REFERENCES

/BeGr 88/ J. Bergin and S. Greenfield, "What Does Modula−2 Need to Fully Support Object Oriented Programming?," *SIGPLAN Notices*, Vol. 23 (3), Mar 88, 73−82.

/BeGr 89/ J. Bergin and S. Greenfield, "What Does Modula−2 Need to Fully Support Object Oriented Programming?," *Journal of Object Oriented Programming,* Vol. 1 (6), Mar/Apr 89, 31−38.

/Blas 89/ G. Blaschek, "Implementation of Objects in Modula−2," *Structured Programming*, Vol. 10 (3), 147−155.

/BrHa 87/ J. Bruckner and J. Harp, "Implementing Opaque Types in Modula−2," *Journal of Pascal, Ada, & Modula−2,* Vol. 6 (4), Jul/Aug 87, 14−30.

/Ca.et.al 88/ L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B.Kalsow, and G. Nelson, "Modula−3 Report," digital Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301.

/Cook 89/ W. Cook, "A Proposal for Making Eiffel Type−safe," *The Computer Journal*, Vol. 32, No. 4, Aug 89, 305−311.

/Craw 87/ A.L. Crawford, "An Extension to Modula−2 For Generic Types," *Journal of Pascal, Ada, & Modula−2,* Vol. 6 (6), Nov/Dec 87, 11−16.

/CzIg 85/ J. Czyzowicz and M. Iglewski, "Implementing Generic Types in Modula−2," *SIGPLAN Notices*, Vol. 20 (12), Dec 85, 26−32.

/ElStr 90/ A. Ellis and B. Strousstrop, "The Annotated C++ Reference Manual", Addison Wesley, 1990.

/GoRo 83/ A. Goldberg and D. Robson, "Smalltalk−80, The Language and its Implementation," Addison−Wesley 1983.

/Goug 86/ K.J. Gough, "Writing Generic Utilities in Modula−2," *Journal of Pascal, Ada, & Modula−2,* Vol. 5 (3), May/Jun 86, 53−62.

/HaWe 89/ D.E. Harms and B.W. Weide, "Types, Copying, and Swapping: Their Influences on the Design of Reusable Software Components," Technical Research Report OSU−CISRC−3/89−TR13, Ohio State University, Columbus, Ohio, Mar 1989.

/HaWe 91/ D.E. Harms and B.W. Weide, "Copying and Swapping: Influences on the Design of Reusable Software Components," *IEEE Transactions on Software Engineering*, Vol. 17 (5), Mai 91, 424−435.

/LeSp 87/ J.J. Leeson and M.L. Spear, "Type Independent Modules: The Preferred Approach to Generic ADTs in Modula−2," *SIGPLAN Notices*, Vol. 22 (3), Mar 87, 65−70.

/Lins 89/ C. Lins, "The Modula−2 Software Component Library, Vol. 1," Springer−Verlag, 1989.

/Mey 88/ B.Meyer, "Oject−Oriented Software Construction," *Prentice Hall,* Englewood Cliffs, NJ (1988)

/Oder 89/ M. Odersky, "Extending Modula−2 for Object−Oriented Programming," *Proceedings of the First International Modula−2 Conference*, Bled, Jugoslavia, Oct 89, 29−35.

/Phil 91/ F. Phillip, "The Support of Genericity in an Editor for Programming−in−the−Large", diploma thesis (in german), Lehrstuhl für Informatik III, RWTH−Aachen, 1991.

/Reyn 87/ C.W. Reynolds, "On Implementing Generic Data Structures in Modula−2," *Journal of Pascal, Ada, & Modula−2,* Vol. 6 (5), Sep/Oct 87, 26−38.

/Wegm 86/ A. Wegmann, "Object−Oriented Programming Using Modula−2," *Journal of Pascal, Ada, & Modula−2,* Vol. 5 (3), May/Jun 86, 5−17.

/Wirth 85/ N. Wirth, "Programming in Modula−2, 3rd ed.," Springer−Verlag, 1985.

/Wirth 88/ N. Wirth, "Type Extensions," *ACM Transactions on Programming Languages and Systems*, Vol. 11 (2), Apr 88, 204−214.

/Wirth 89/ N. Wirth, "Modula−2 and Object−Oriented Programming," *Proceedings of the First International Modula−2 Conference*, Bled, Jugoslavia, Oct 89, 7−12.

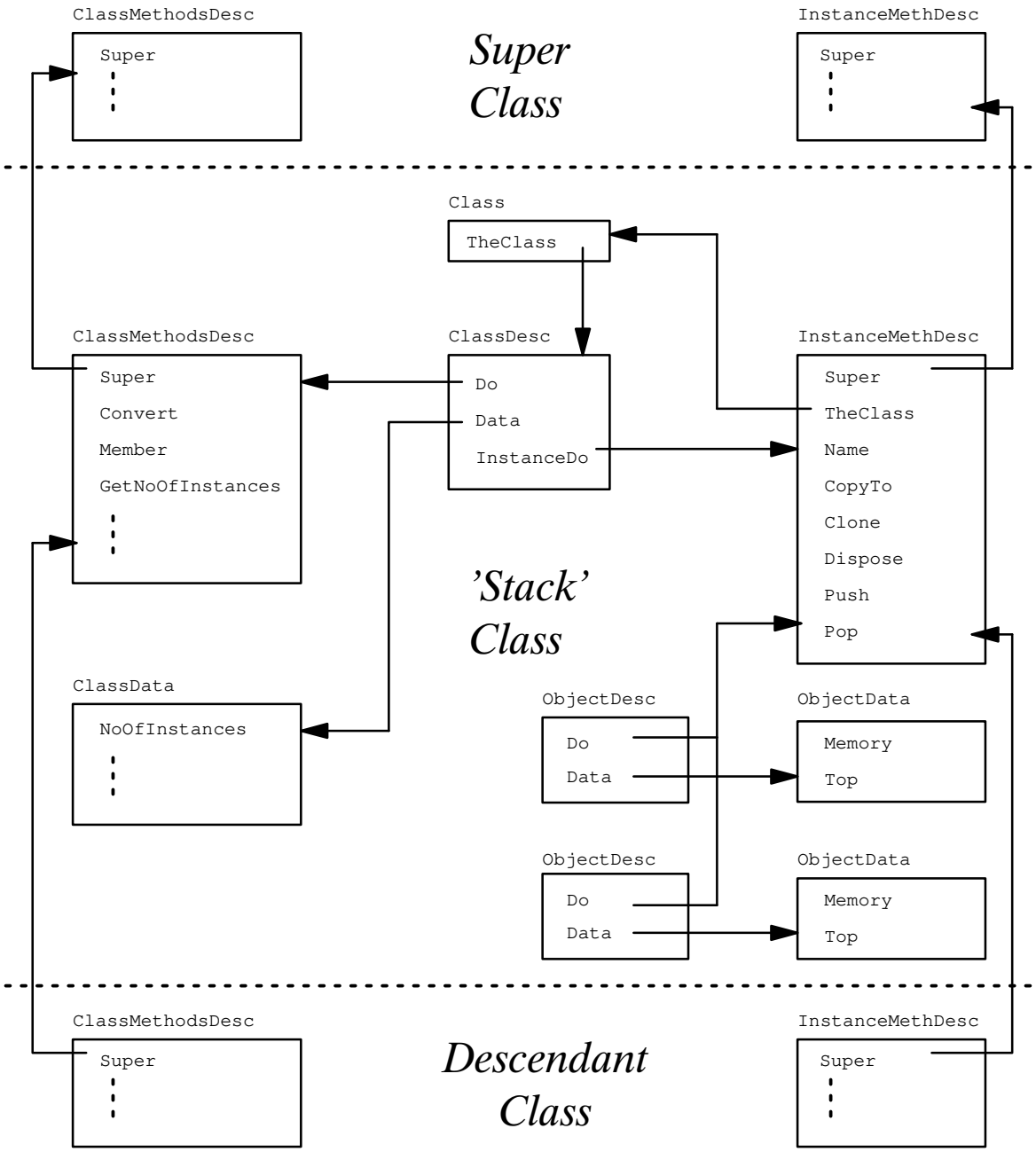
/Wien 84/ R.S. Wiener, "Generic Sorting in Modula−2," *Journal of Pascal, Ada, & Modula−2,* Vol. 3 (1), Jan/ Feb 84, 33−36.

/WiSi 85/ R.S. Wiener and R.F. Sincovec, "Two Approaches to Implementing Generic Data Structures in Modula−2," *SIGPLAN Notices*, Vol. 20 (6), Jun 85, 56−64.

/Zimm 85/ J.A. Zimmer, "A Modest Modula Wish List," *SIGPLAN Notices*, Vol. 20 (11), Nov 85, 69−77.

Appendix A: The used Data Structures

The following picture shows the data structure for a (not deferred) class `Stack`. Two objects of this class have already been created. The references to them are not contained.



The arrows in this picture represent references that will be constructed at program start or when an object of that class is created. Every box represents a record allocated on the program's heap. The names above the boxes denote the type of that object. This picture corresponds to the following Modula−2 modules.

```
DEFINITION MODULE Stack; (* IS_A ANY *)
   FROM SYSTEM IMPORT ADDRESS;
   FROM GlobalTypes IMPORT ComplexNumber;

   IMPORT ANY;

   CONST MaxStackSize = 10;

   TYPE
      ClassName              = ARRAY [ 1 .. 20 ] OF CHAR;
      Object                 = POINTER TO ObjectDesc;
```

```
      ObjectData;
      Class                  = POINTER TO ClassDesc;
      ClassData;
      InstanceMethods        = POINTER TO InstanceMethodsDesc;

      (* Inherited instance method types *)
      CopyToProcType         = PROCEDURE( Object, Object );
      CloneProcType          = PROCEDURE( Object ) : Object;
      DisposeProcType        = PROCEDURE( Object );
      DisplayDataProcType    = PROCEDURE( Object );

      (* New defined instance method types *)
      PushProcType           = PROCEDURE( Object, ComplexNumber);
      PopProcType            = PROCEDURE( Object) : ComplexNumber;

      InstanceMethodsDesc    = RECORD
                                  Super    : ANY.InstanceMethods;
                                  TheClass : Class;

                                  (* class name required for messages *)
                                  Name     : ClassName;

                                  (* Inherited instance methods *)
                                  CopyTo   : CopyToProcType;
                                  Clone    : CloneProcType;
                                  Dispose  : DisposeProcType;

                                  (* New defined instance methods *)
                  (* DEFERRED *)  Push     : PushProcType;
                  (* DEFERRED *)  Pop      : PopProcType;
                               END (* record *);

      ObjectDesc             = RECORD
                                  Do   : InstanceMethods;
                                  Data : ObjectData;
                               END (* record *);

      ClassMethods           = POINTER TO ClassMethodsDesc;

      (* Inherited class method types *)
      ConvertProcType        = PROCEDURE( Class, ADDRESS) : Object;
      MemberProcType         = PROCEDURE( Class, ADDRESS) : BOOLEAN;

      (* New defined class methods *)
      GetNoOfInstancesProcType = PROCEDURE( Class) : CARDINAL;

      ClassMethodsDesc       = RECORD
                                  Super             : ANY.ClassMethods;

                                  (* Inherited class methods *)
                                  Convert           : ConvertProcType;
                                  Member            : MemberProcType;

                                  (* New defined class methods *)
                                  GetNoOfInstances  : GetNoOfInstancesProcType;
                               END (* record *);

      ClassDesc              = RECORD
                                  Do                : ClassMethods;

                                  (* encapsulated classdata *)
                                  Data              : ClassData;

                                  (* instance methods part *)
                                  InstanceDo        : InstanceMethods;
                               END (* record *);

   (* Standard procedures *)
   PROCEDURE CONSTRUCTOR( Self : Object);
   (* class hierarchy entry for this class *)
   VAR
      TheClass       : Class;
END Stack.

(*************************************************************************)
IMPLEMENTATION MODULE Stack;
   FROM SYSTEM IMPORT ADDRESS, TSIZE;
   FROM Storage IMPORT ALLOCATE, DEALLOCATE;
   FROM InOut IMPORT WriteLn, WriteString;
   FROM GlobalTypes IMPORT ComplexNumber;

   IMPORT ANY;

   TYPE
      ClassData  = POINTER TO RECORD
                      (* Inherited class data *)
                      (* New defined class data *)
                      NoOfInstances : CARDINAL;
```

```
                  END (* record *);
      ObjectData = POINTER TO RECORD
                      (* Inherited instance data *)
                      (* New defined instance data *)
                      Dummy : CARDINAL;
                   END (* record *);
   (* Standard procedures ------------------------------------------------ *)
   PROCEDURE CONSTRUCTOR( Self : Object );
   BEGIN
      (* First call the CONSTRUCTOR of the super class *)
      ANY.CONSTRUCTOR( TheClass^.Do^.Super^.Convert( ANY.Class( TheClass ), Self ) );
      (* Because there's no data, there's no initialization provided here *)
   END CONSTRUCTOR;
   (* New defined instance methods --------------------------------------- *)
   (* DEFERRED *)
   PROCEDURE PushProc( Self : Object; SomeNumber : ComplexNumber );
   BEGIN
      ANY.SuperClassError( "Push", Self );
   END PushProc;
   (* DEFERRED *)
   PROCEDURE PopProc( Self : Object ) : ComplexNumber;
   BEGIN
      ANY.SuperClassError( "Pop", Self );
   END PopProc;
   (* New defined class methods ------------------------------------------ *)

BEGIN (* ------------------------------------------------------------------ *)
   NEW( TheClass );
   WITH TheClass^ DO
      (* Initialization of the class methods *)
      NEW( Do );
      WITH Do^ DO
         Super     := ANY.TheClass^.Do;
         (* Inherited class methods *)
         Convert   := ConvertProcType( Super^.Convert );
         Member    := MemberProcType( Super^.Member );
      END (* with *);
      (* Initialization of the class data *)
      NEW( Data );
      WITH Data^ DO
         NoOfInstances := 0;
      END (* with *);
      (* Initialization of the instance methods *)
      NEW( InstanceDo );
      WITH InstanceDo^ DO
         Super     := ANY.TheClass^.InstanceDo;
         Name      := "Stack";
         (* Inherited instance methods *)
         CopyTo    := CopyToProcType( Super^.CopyTo );
         Clone     := CloneProcType( Super^.Clone );
         Dispose   := DisposeProcType( Super^.Dispose );
         (* New defined instance methods *)
         Push      := PushProc;
         Pop       := PopProc;
      END (* with *);
      InstanceDo^.TheClass := TheClass;
   END (* with *);
END Stack.
```