Science & Technology Facilities Council

Technical Report

State of the Art in Object Oriented Programming with Fortran

D. J. Worth

January 4, 2008

RAL-TR-2008-002

Library and Information Services
SFTC Rutherford Appleton Laboratory
Harwell Science and Innovation Campus
Didcot
OX11 0QX
UK
Tel: +44 (0)1235 445384
Fax: +44(0)1235 446403
Email: library@rl.ac.uk

ISSN 1358-6254

Abstract

To define the "state of the art" in object oriented (OO) programming with Fortran we bring together a number of disparate resources and try to distill from them recommendations for an OO approach. We begin by defining what the author means by "object oriented" and give details of the particular context in which this paper is set. With this background in place we present a survey of tools and techniques (including the Fortran 2003 standard) that allow Fortran programmers to develop their software in an object oriented way. The thorny question of whether efficiency *must* be sacrificed with object orientation is tackled later in the paper showing that some key elements of OO design can be used successfully with the right compiler.

1 Introduction

Fortran has been the language of choice for scientific programmers for 50 years, primarily because it handles arrays well and compilers produce code that runs quickly. During Fortran's life many new programming languages have come (and some gone) and new software engineering methods have been introduced. Most Fortran programmers know and love the procedural design methodology the language brings with it, and can produce well structured codes that are easy to maintain. In the last 20 years the notion of object oriented (OO) design has taken off in the world of commercial software engineering with its promises of short development time, flexible architectures and code reuse. The Fortran standard has developed from Fortran 77 through 90, 95 and now to 2003 and at each revision has added more object oriented concepts to the language. Fortran programmers however have not been so quick to take these new features on board in their coding. There are a number of reasons for this, the most prominent and powerful being the amount of legacy code that exists that it would be too expensive in time and money to transform. The author knows of one project that re-wrote their software from scratch to make use of new features in Fortran 90 but this is a very rare event.

So what are new Fortran programmers to do? Should they ignore the object oriented features and continue procedural programming or should they start on a process to transform their code to this new design paradigm? The answer of course is a project management decision but in this document we will present the "state of the art" in object oriented programming with Fortran and suggest ways in which it can be used to improve the quality and maintainability of software.

This report cannot go in to details on object oriented design techniques - the web would be a better place to start - but we do give a brief description of the concepts of OO design and the benefits that may accrue from using them in Section 2. Following this we present techniques and tools for object oriented programming in Fortran in Section 3 and then consider the effect of data hiding on efficiency in Section 4. Some recommendations on the use of OO programming conclude the report.

There are code fragments throughout the report and a set of appendices with more complete examples can be found at the back of the report.

2 What is "Object Oriented"?

There are many many definitions of "object oriented" in software engineering referring to software *analysis and design* and *implementation*. Analysis and design is the most important area in which to apply object oriented techniques as it will lead naturally to implementing software in an object oriented way. There are many books on object oriented analysis and design, one which gives a good introduction and includes material on OO analysis and design with non-OO languages is [1]. Other references are [2], [3] and [4] and a web search for "object oriented design" will reveal good introductory pages.

Both design and implementation need to be addressed for software to be considered object oriented and we make it clear from the outset that the implementation language *does not* define whether software is OO, as it is possible to write procedural code in C++, Java or any other language that supports objects.

The references cited above will give good definitions of "object oriented" but for the purposes

of this report we will give a quick summary of the concepts involved in OO here.

Classification This is the use of domain specific knowledge to define the "objects" from the domain in which the software operates. Objects contain *data* that describe or are properties of the object and *methods* (functions) to act on this data. Examples could include:

- A sparse matrix in which the row, column and value are stored for non-zero elements along with the dimensions of the matrix. Methods could include access to elements via row and column, matrix-vector and matrix-matrix multiplication, factorisation and possibly solution of $Ax = b$.
- A radioactive element that has a name and half life and knows the names of its daughter(s).

Objects are often implemented as classes in OO languages and the terms are interchangeable in non-technical discussion of OO.

Encapsulation The objects we wish to use will store their data in a particular way and the idea of encapsulation is to hide this choice. Methods are provided for other objects to use to access data and perform operations on the data. These methods define the *interface* of the object. The methods use inside knowledge to be as efficient as possible and also hide details of the data storage. This means that the details can be changed or methods improved without impacting other objects.

Inheritance This allows objects to share data and methods among a group of related objects and is commonly achieved in one of the following two ways:

"is-a" For example a `Student` object may inherit from `Person` because a student *is a* person with name, date of birth, sex etc. but adds more information such as subjects studied and results. This is also known as *public inheritance*.

"has-a" For example a `Molecule` object may use an `Atom` object because a molecule *has a* (is composed of) a number of atoms. This is also known as *delegation*.

In both cases existing code is reused with consequent saving in implementation and maintenance time.

Polymorphism The concept of polymorphism comes in two strands.

static polymorphism In this case the type of the object is known at compile time and the correct method is called depending on the type and objects can overload methods. This is already in Fortran 95 via a generic interface in a module.

run-time polymorphism This follows from the idea of inheritance. Sub-objects can override methods on the base objects with methods of their own (with the same signature) and the particular method called will be chosen at run time based on the object type. This means we can write code that works on the base object using base object methods and know that if the code comes across a sub-object it will call the correct method on the sub-object. (This is also known as *dynamic dispatch*.)

For example: circle, square and triangle could all inherit a method from shape to translate them in 2D. The implementation would be different in each case (because the data in each class are different) but code written using shape would work correctly when given a circle, square or triangle.

That will do for the purposes of this document but much more could be included, though such material is often language dependent. The four facets given above are enough to take forward into the next section which looks at object oriented programming with Fortran 95.

3 A Survey of Techniques and Tools

Many languages (e.g. C++ and Java) have been designed with object oriented programming in mind and as a consequence provide inheritance and polymorphism as part of the language. Currently Fortran does not - in fact it lacks the concept of methods being bound to objects although this has been changed in the Fortran 2003 standard, see [5].

So if we want to use OO features then we must code them ourselves. In the following sections we discuss how this may be done and take a look at tools that may help simplify or reduce the work.

Our focus is on implementation and that being so we address the OO concepts of *encapsulation*, *inheritance* and *polymorphism*. The remaining concept, *classification*, is a simple one to grasp and so we will not go into it any further except to say that it involves knowledge of the problem domain and that it provides our starting point – a design comprising objects and their interfaces.

3.1 Techniques

There are many references concerning object oriented development with Fortran (the ACM electronic library lists 200). Many are concerned with explaining how to implement object-oriented features in Fortran 95 [6, 7, 8, 9, 10, 11, 12, 13] and then others report on software developed in an OO manner with Fortran [14, 15, 16, 17, 18]. The work of Decyk, Norton and Szymanski is available via the oft cited `http://www.cs.rpi.edu/~szymansk/oof90.html`. The ideas presented in the following sections are based on these references.

3.1.1 Encapsulation

Our object oriented design contains objects (classes) and it is obvious that we are going to represent them with Fortran modules. The module will in all likelihood contain a `type` definition that encapsulates the data of the object which means that the object can be passed easily in sub-program calls. The methods of the object will be `contain`ed in the module.

It would be possible to make the components of the data type `private` and provide functions and subroutines in the module to get and set the data. These sub-programs define the interface to the data and separate the user of the data from details of how data is stored allowing changes to the storage without affecting the user. This "data hiding" may be over the top for some but others will consider it worthwhile. We will see in Section 4 that the correct choice of compiler and options eliminates any overhead of using the interface.

3.1.2 Inheritance

In Fortran 95 there is no language feature that allows one object to "extend" another to implement "is-a" inheritance. We are left with what is technically delegation – one object using another – to cover both forms of inheritance.

One example of extending one class to another is given in [11] using the person/student example. The person object (with identification number, first and last names) is implemented in a module as follows (omitting the methods)

```
module Personnel_class

    type Personnel
        integer :: ssn
        character*12 :: firstname, lastname
    end type Personnel

    contains

    ! Methods omitted for this discussion

end module Personnel_class
```

The student object extends person with the subjects (classes) that it takes as follows:

```
module Student_class

    ! Bring Personnel_class into scope
    use Personnel_class

    ! Define the student type
    type Student
        private
        type(Personnel) :: personnel
        integer :: nclasses
        character*12, dimension(10) :: classes
    end type Student

    contains

    ! Methods omitted for this discussion

end module Student_class
```

Note here three things

- The data for each object is stored in a derived type defined within a module.

- The module will also contain the methods so that the data and methods can be brought into scope together.
- The `Student` type defines all its data as private and so its containing module must provide methods to access the data and data from its associated `Personnel` object.

3.1.3 Run-time Polymorphism

We defined run-time polymorphism as the ability to call the correct method based on the type of an object when that type is only known during execution. As an example consider the shape classes in Appendix A. We define a `Shape_class` base class for `Circle_class` and `Square_class` subclasses. These subclasses have "constructors" (the `new`) and methods to move and print out their data. The `interface` mechanism provides some run-time polymorphism, for example we can call `print` on a circle or square and the correct subroutine will be executed.

To achieve true run-time polymorphism we need an object which can represent either a circle or square when used in a subroutine, be assigned to either of these types, and know which actual type it represents so the correct routine can be called at run-time. This is implemented in the `poly_Shape_class` module which contains a `poly_Shape` type that has a pointer to each of the `Circle` and `Square` objects only one of which will be associated at a time. The interfaces in the module defines methods to initialise one of these polymorphic objects (both pointers nullified), `new`; assign a circle or square to a polymorphic object, `poly`; and implement dynamic dispatch for methods on the subclasses, `print` and `move`.

Assignment of a square object to the polymorphic object sets the `pSquare` pointer and nullifies the `pCircle` pointer. Assignment of a circle object is similar. Dynamic dispatch is then implemented by checking the `associated` status of each pointer and calling the method on the associated one.

This implementation is based on the student/teacher/database example in [11]. A second, simplified implementation is described by the same authors in [12]. Instead of creating separate types for each of the polymorphic objects they propose creating a single type to hold the combined data of every object plus a flag to determine exactly which object the type is representing. This amalgamated type can be thought of as a base object whose methods other classes can override or add to. We implement constructors for the various objects (defined in their own modules) that set the flag appropriately along with the overriding or additional methods for those objects. A separate polymorphic type is still implemented that defines interfaces and implementations for the polymorphic functions (and only those functions) using the flag to determine which actual routine to call. A full description of this method can be found in [12].

The advantage of this second implementation is that only the methods that override methods in the base, amalgamated object need have a polymorphic implementation in the polymorphic object since we could just call the method on the base class if it was not modified. The first implementation requires that all methods to be used polymorphically have an implementation in the polymorphic object. The disadvantage however is that encapsulation of data in separate objects has been lost and extending the inheritance hierarchy with new objects becomes more difficult. Not because there is more coding, simply because it is more difficult to see what the designed inheritance hierarchy was that gave rise to the amalgamated object.

We feel that this disadvantage outweighs the advantages especially in the light of the Forpedo

tool introduced in the next section.

3.2 Tools

The previous section shows what can be done. The next question is "Are there any tools to help us?"

3.2.1 Forpedo

Forpedo (`http://www.macanics.net/forpedo/index.php`) is a preprocessor for Fortran 95 code and supports templates and polymorphism as well as more typical preprocessing tasks. Templates are used widely in C++ and allow programmers to write a class that operates on a generic object and then create such a class for a specific object without having to re-create the code for each particular object. This is also known as *generic programming*. For example we could write a linked list module for a generic object and use it to store integers, real numbers or other user defined types. Templates are built in to C++ but not in to Fortran and this is where Forpedo comes in. By writing modules that use some generic name enclosed by '<' and '>' and defining the various values for the generic object Forpedo can create code for each value. Forpedo is written in Python. A small example of templates taken from [19] is given in Appendix B.

As we saw in Section 3.1.3 the implementation of polymorphism is complex in Fortran and requires a great deal of coding to get working. Forpedo seeks to reduce the effort by writing the polymorphic dispatch code for us in a superclass module. The user has to write the code for each of the modules defining the concrete polymorphic types (subclasses) and special code that defines the superclass. This special code is processed by Forpedo to produce Fortran 95 code for the superclass containing the dynamic dispatch code. This module can then be used in subprograms that know nothing about the concrete types. An example from the web site is given in Appendix C.

One point to note here is that the superclass module defines the assignment operator so that subclass objects can easily be assigned to superclass objects for passing to routines written only in terms of the superclass.

3.2.2 Ideas from the CÆSAR Code

The CÆSAR code package (`http://www.lanl.gov/Caesar/Caesar.html`) is a computational physics environment allowing users to model the physics of real systems by solving systems of partial differential equations. Its main focus is on equation sets, discretisation on meshes, nonlinear solvers and preconditioners, leaving linear solvers, mesh generation and partitioning, communication and visualisation to external packages. CÆSAR is written in Fortran 90 and makes use of `m4` macros in the preprocessing stage.

One set of macros is the "superclass" set and this is used to automatically create a superclass module to take care of the dynamic dispatching of function calls among a set of subclasses. Details are given in Section 6.5 of [20] and summarised below.

- Define the subclass modules (each must be called `subclassname_Class`) and the subclass types contained therein (each must be called `subclassname_type`).
- Define functions and subroutines in the subclasses (each must be called `routinename_subclassname`).

- Define interfaces in the subclasses for the functions and subroutines (each must be called `routinename`).
- Define the superclass module using the "superclass" `m4` macros.
- Create the Fortran code for the superclass module by running `m4` on the definition from above.

This is not such a neat solution as Forpedo, partly because of the naming and coding conventions but also for the lack of the simple assignment from subclass to superclass available in Forpedo.

3.3 Fortran 2003

In the previous sections we have mentioned features of Fortran 95 that enable object oriented programming, for example the use of interfaces to achieve compile-time polymorphism, and we have introduced techniques to emulate inheritance and run-time polymorphism. In the new Fortran standard (Fortran 2003) features have been introduced to enhance the object oriented nature of the language. An excellent review of the new features of Fortran 2003 is provided in [5] and we provide details of the particular OO features in this section. The latest "Fortran Explained" book [21] also includes a chapter on object oriented programming.

3.3.1 Type extension

We have shown above how inheritance among objects may be implemented in Fortran 95 but Fortran 2003 includes inheritance as a language feature. One type may *extend* another, its *[parent* (so long as the parent does not have the `SEQUENCE` or `BIND(C)` attribute). For example the circle object in Section 3.1.3 may be defined to extend the shape object as follows.

```
! Define Circle type
type, extends(Shape) :: Circle
    private
    real :: centre_x, centre_y    ! centre coordinates
    real :: radius                ! radius
end type ! Circle
```

All the type parameters, components and bound procedures (see later) of the parent type and are known by the same names. They may be accessed using the component selection syntax, `%`. The whole of a parent type may be accessed using its name, for example with `type(Circle) :: c` we can access the `Shape` part by `c%Shape`.

There is an assumed order of variables for structure constructors and for input/output. It consists of inherited components in component order of the parent type followed by new components.

Another new feature is increased control of access to components of derived types. Individual components may be declared `PUBLIC` or `PRIVATE`, including procedures bound to types (see next sections).

3.3.2 Procedure pointers

A pointer variable, whether a component of a derived type or not may now be a procedure pointer. The declaration

```
PROCEDURE (proc), POINTER :: p => NULL()
```

defines the pointer `p` to be a procedure pointer, initially null and having the same interface as the procedure `proc`. If there is no procedure to act as the interface template then an *abstract interface* can be defined without there being an actual function with that interface. The interface template may be omitted and the function pointer has an implicit interface and may be associated with a function or subroutine. Alternatively the template may be replaced by a function return type and then the pointer may only be associated with a function with that return type.

The addition of procedure pointers gives Fortran programmers the ability to bind procedures to derived types allowing methods to be carried with objects. This is not the typical OO language idea of methods since a function pointer may be associated with different functions throughout the lifetime of the object, whereas methods remain the same. The latter situation is catered for by binding a function to a type described in the next section.

3.3.3 Procedures bound by name to a type

A procedure may now be `contain`ed in a derived type and be accessed from an instance of that type by the component selection syntax. Several of these bound procedures can be made available via a single generic name using the `GENERIC` keyword, for example

```
GENERIC :: gen => proc1, proc2, proc3
```

where `proc1`, `proc2` and `proc3` are names of specific procedures. The disambiguation of the procedures follows the usual rules.

Usually a procedure bound to a derived type in this way requires access to data contained in the instance of the type. The default is to assume that the instance is passed as the first argument and the other arguments move along. In this case the procedure is said to have the `PASS` attribute (this is the default but may be explicitly confirmed in the procedure declaration). If this behaviour is not required the procedure must be declared with the `NOPASS` attribute.

The procedures may be bound to a type as an operator or defined assignment and uses the `GENERIC` statement to provide the operators and assignment for various contexts.

3.3.4 Procedure overriding

A procedure bound to an object that extends another may have the same name and attributes as a procedure bound to the parent. Such a procedure must not be PRIVATE if the procedure on the parent is PUBLIC. We can stop the overriding of a procedure by declaring it as follows:

```
PROCEDURE, NON_OVERRIDABLE :: proc
```

3.3.5 Finalization

With derived types now looking more like objects with data and methods contained in them the new standard has introduced the idea of final subroutines. Their purpose is to clean up when the type ceases to exist, for example to deallocate targets of pointer components or close a file. Final subroutines must be module procedures and take a single argument which is the object about to be destroyed. The syntax for declaring module subroutines to be final is

```
type T
    : ! Component declarations
contains
    FINAL :: finish1, finish2
end type ! T
```

The final subroutines form a generic set and follow the usual rules for disambiguation. Each one will be used for different ranks of variables of type T or for variables of different kind type parameter (see next section).

When a finalizable derived type `T1` has a component of another finalizable derived type (`T2`) the finalization subroutine is called for `T1` first followed by finalization for `T2`. This means that the final subroutine of an object only has to concern itself with the object's components that are not finalizable.

The final subroutines of a type are not inherited by types that extend it, although they are run when an instance of the extended type ceases to exist. Final subroutines can be implemented for the extended type and they are invoked before the final subroutine of the parent type.

3.3.6 Parameterised derived types

Derived types may be parameterised by kind and length parameters. The parameterisation by kind is akin to the idea of templates in C++ which can currently be implemented using Forpedo. The kind parameter is fixed at compile time and may be used as the kind in any component of the derived type (including another parameterised derived type).

3.3.7 Polymorphic entries

We have seen above how Fortran 2003 has made object oriented programming easier, especially the construction of inheritance hierarchies. All that remains is a way to define variables able to represent the base class or any of its extensions during execution. These are known as polymorphic entries. We can define such object by using the `CLASS` keyword in place of the `TYPE` keyword in the declaration. The entity must have the pointer or allocatable attribute or be a dummy argument. It will get its type via pointer assignment, allocation or argument association at run-time.

Now we can write code for objects of one type and have them work for objects that extend this type automatically. Within such code we will have access only to type parameters, components and bound procedures of the declared type. This is as it should be – we are writing code to manipulate the base object so we should only need information about that object. Strictly, if we wanted to do something with an extension of that object we should write different code. However this is not always the best option so the `SELECT TYPE` construct has been introduced to select

one of its blocks of code to be executed depending on the type of a variable or an expression. An example of this with the Shape and Circle types introduced in Section 3.1.3 is given below.

```
TYPE(Shape), TARGET :: s
TYPE(Circle), TARGET :: c
CLASS(Shape), POINTER :: p

p => c

SELECT TYPE( p )
TYPE IS (Circle)
    print *, 'This is a circle'
CLASS IS (Shape)
    print *, 'This is a shape'
END SELECT
```

The block to be executed is chosen according to the following rules:

1. If a `TYPE IS` block matches, it is taken.
2. Otherwise, if a single `CLASS IS` block matches, it is taken.
3. Otherwise, if several `CLASS IS` blocks match one must be an extension of all the others and it is taken.
4. Otherwise, take the `CLASS DEFAULT` block, if it is present.

3.3.8 Deferred bindings and abstract types

There are times when we want to define a base object only as a template for other objects, specifying the methods the extensions must implement but not implementing them itself. For example code using the Shape object seen previously may want each shape to be able to report its area and so we would want to require that every type that extends Shape must have a `getArea` function. We can do this by defining an abstract type with abstract interface as follows.

```
type, abstract :: Shape
    contains
        procedure (calculate_area), deferred, pass :: getArea
        ...
end type ! Shape

abstract interface
    function calculate_area(theShape)
        real :: calculate_area
        class(Shape), intent(inout) :: theShape
    end function ! calculate_area
end interface
```

Note that polymorphic entries may be declared (using `CLASS`) to be of an abstract type but it is not permitted to declare, allocate or construct a non-polymorphic object of such a type. Also deferred binding is only allowed in an abstract type.

4 Efficiency, Efficiency, Efficiency

There is much opposition to object oriented programming within the computational science community and the objections are three-fold.

- We have a great deal of legacy code (mostly Fortran) and cannot spare the effort to rewrite it in some other OO language.
- We have a great deal of legacy code and cannot spare the effort to rewrite it in Fortran 95.
- We cannot and must never ever ever let anything get in the way of performance.

The first objection can be accepted but is a spurious reason for not improving the design of a Fortran code because we can do many good things with Fortran 95. Modularisation and derived data types can be used to improve code – it is not always straightforward (hence the second objection) but neither is it impossible, [22, 23, 24].

So we come to the major stumbling block – efficiency. Surely all that data hiding and modularisation eats up cycles? Well that may be so for some implementations, but we are not advocating object orientation for its own sake. The techniques shown above can be useful and, as we will see, do not impact efficiency. What if a good design made it easier to change linear solvers when a better one came along? There's a benefit not to be sniffed at!

Data hiding looks like a problem as far as efficiency is concerned. Lots of function/subroutine call overhead just to get at data? We conducted an experiment to compare the use of public and private data in derived types using different compilers and their options.

The simple code used is given below. There are two types, `PrivateTestType` that declares its data `private` and provides functions for access, and `PublicTestType` in which the data is freely accessible.

```
module TestTypes

  use precision_module, only : WP

  implicit none

  ! A type that hides data
  type PrivateTestType
    private
      integer :: intValue
      real(kind=WP) :: realValue
  end type ! PrivateTestType

  ! A type that has public data
```

```
  type PublicTestType
    integer :: intValue
    real(kind=WP) :: realValue
  end type ! PublicTestType

  contains

    subroutine setIntValue(object, value)
      type(PrivateTestType), intent(inout) :: object
      integer, intent(in) :: value
      object%intValue = value
    end subroutine ! setIntValue

    subroutine setRealValue(object, value)
      type(PrivateTestType), intent(inout) :: object
      real(kind=WP), intent(in) :: value
      object%realValue = value
    end subroutine ! setRealValue

    function getIntValue(object)
      integer :: getIntValue
      type(PrivateTestType), intent(in) :: object
      getIntValue = object%intValue
    end function ! getIntValue

    function getRealValue(object)
      real(kind=WP) :: getRealValue
      type(PrivateTestType), intent(in) :: object
      getRealValue = object%realValue
    end function ! getRealValue

end module ! TestTypes
```

We wrote two tests of interest in order to investigate the overheads of data hiding.

1. Does wrapping raw data in a derived type (with public components) affect performance? Perform a calculation involving real and integer data 10^7 times using a subroutine that takes a derived type containing the data and using a subroutine where the raw data are supplied.

2. Does making components of a derived type private (meaning we must use procedures to set and get the data) affect performance? Get and set the data for a `PrivateTestType` instance 10^7 times and do the same for a `PublicTestType` instance.

The results are given in Table 1.

	Test 1		Test 2	
	Derived	Raw	Private	Public
g95 – no optimisation	0.26	0.27	0.34	0.12
`-O2`	0.26	0.25	0.21	0.01
`-O3`	0.25	0.27	0.20	0.02
`-march=nocona -ffast-math -funroll-loops -O3`	0.09	0.1	0.16	0.02
NAG – no optimisation	0.26	0.26	0.43	0.36
`-O2`	0.29	0.51	0.23	0.03
`-O4`	0.28	0.49	0.21	0.03
`-O4 -mismatch_all -ieee=full -Bstatic`	0.26	0.51	0.19	0.03
Intel – no optimisation (-O0)	0.29	0.29	0.38	0.12
`-O2`	0.19	0.22	0.19	0.02
`-O3`	0.23	0.23	0.19	0.01
`-ipo -O3`	0.22	0.23	**0.02**	**0.02**
Lahey – no optimisation	0.23	0.23	0.26	0.02
`-O`	0.22	0.23	0.27	0.02
`--o2`	0.26	0.26	0.26	0.02
`-x - --o2`	0.05	0.03	0.26	0.01

Table 1: Table of results for overhead tests 1 and 2

The table above shows that there need be no overhead in using derived types, a fact enhanced by noting that this test included time for setting values of the derived type's components before each call to the subroutine.

The emboldened line in the table shows that by inlining the set and get routines the Intel compiler can deliver the same performance for encapsulated data as for publicly accessible data in a derived type.

Test 2 described above can be considered a worst-case of data-hiding in that we are considering a set of 10^7 objects individually. A better approach would be to consider on object that describes the collection of these individuals. For example, it may be better to create an object that describes a molecule rather than deal with an a group of individual atom objects. To test this we created Test 3 that repeated Test 2 with objects that contained array components which were set and get as appropriate for their public/private nature. The module we used is given below.

```
module ArrayTest

  use precision_module, only : WP

  implicit none

  ! A type that hides data
  type PrivateArrayTestType
    private
      integer, pointer, dimension(:) :: intArray
      real(kind=WP), pointer, dimension(:) :: realArray
```

```
end type ! PrivateArrayTestType

! A type that has public data
type PublicArrayTestType
  integer, pointer, dimension(:) :: intArray
  real(kind=WP), pointer, dimension(:) :: realArray
end type ! PublicArrayTestType

contains

  subroutine setIntArray(object, Array)
    type(PrivateArrayTestType), intent(inout) :: object
    integer, target, dimension(:), intent(in) :: Array
    object%intArray => Array
  end subroutine ! setIntArray

  subroutine setRealArray(object, Array)
    type(PrivateArrayTestType), intent(inout) :: object
    real(kind=WP), target, dimension(:), intent(in) :: Array
    object%realArray => Array
  end subroutine ! setRealArray

  function getIntArray(object)
    type(PrivateArrayTestType), intent(in) :: object
    integer, pointer, dimension(:) :: getIntArray
    getIntArray => object%intArray
  end function ! getIntArray

  function getRealArray(object)
    type(PrivateArrayTestType), intent(in) :: object
    real(kind=WP), pointer, dimension(:) :: getRealArray
    getRealArray => object%realArray
  end function ! getRealArray

  function getIntArraySize(object)
      type(PrivateArrayTestType), intent(in) :: object
      integer :: getIntArraySize
      getIntArraySize = size(object%intArray)
  end function ! getIntArraySize

  function getRealArraySize(object)
      type(PrivateArrayTestType), intent(in) :: object
      integer :: getRealArraySize
       getRealArraySize = size(object%realArray)
```

```
    end function ! getRealArraySize

end module ! ArrayTest
```

The results are given in Table 2.

	Test 3	
	Private	Public
g95 – no optimisation	0.24	0.14
`-O2`	0.24	0.04
`-O3`	0.26	0.03
`-march=nocona -ffast-math -funroll-loops -O3`	0.24	0.03
NAG – no optimisation	0.11	0.11
`-O2`	0.03	0.03
`-O4`	0.05	0.03
`-O4 -mismatch_all -ieee=full -Bstatic`	0.04	0.03
Intel – no optimisation (-O0)	0.15	0.13
`-O2`	0.05	0.04
`-O3`	0.03	0.04
`-ipo -O3`	0.04	0.04
Lahey – no optimisation	0.04	0.03
`-O`	0.04	0.04
`--o2`	0.04	0.03
`-x - --o2`	0.03	0.03

Table 2: Table of results for overhead Test 3

We can see from this table that not only have times come down compared with Test 2 but that there is little difference between using public and private components in these derived types.

5 Recommendations

Be pragmatic. We are not advocating the use of objects and object oriented programming everywhere and in all cases. Indeed there are those who argue that OO programming is not as "natural" as some of its advocates propose [25, 26].

The (somewhat obvious) idea is to use it where it may be useful. Some situations where it may be useful:

- Data is passed among procedures in a group. Use a derived type to make the grouping clear (and shorten argument lists!)
- Data is passed among a small set of procedures in a group. Use a derived type defined in a module along with the procedures. The relation between the data and procedures is made more obvious.
- A set of derived types have data in common. Create a base type containing the common data and use "has-a" inheritance for the derived types. The common data is in one place and not replicated in each type.

- There is a set of routines each doing a similar thing to different types of object plus a large amount of decision code. Put the routines in a module and create a generic interface. The decision about which routine to call is now made by the compiler.
- There are routines all doing the same thing to different derived types. If it really is the same thing then there must be some commonality between the types so create a base class for the derived types and put one copy of the routine in its module that operates on the base class type. The base class data can be retrieved from the derived types and passed to this one routine.
- If there really is polymorphism use polymorphic objects and Forpedo.

We should also add the two main pitfalls to be aware of with OO programming.

- Avoid making the inheritance hierarchy too deep. Object names in very deep hierarchies can become meaningless, e.g. `Object` or `Widget`. It may also make it difficult to fit a new object into the hierarchy, see [26].
- Don't create objects just for the sake of it. Unless you are very good at describing what is going on and why, the purpose of an object may become obscured.

References

[1] J. Rumbaugh *et al. Object-Oriented Modeling and Design.* Prentice Hall International, 1991.

[2] G. Booch. *Object-Oriented Analysis and Design With Applications.* Benjamin/Cummings, 1994.

[3] J. Arlow *et al. UML and the Unified Process : Practical Object-Oriented Analysis and Design.* Addison Wesley, 2002.

[4] S. Bennett *et al. Object-Oriented Systems Analysis and Design Using UML.* McGraw-Hill, 1999.

[5] J.K. Reid. The new features of Fortran 2003. Technical Report ISO/IEC JTC1/SC22/WG5 N1648, 2004.

[6] B.J. Dupée. Object oriented methods using Fortran 90. *ACM SIGPLAN Fortran Forum*, 13(1):21, 1994.

[7] V.K. Decyk, C.D. Norton, and B.K. Szymanski. Introduction to object-oriented concepts using Fortran90. Technical Report UCLA IPFR Report PPG-1560, 1996.

[8] V.K. Decyk, C.D. Norton, and B.K. Szymanski. Expressing object-oriented concepts in Fortran 90. *SIGPLAN Fortran Forum*, 16(1):13–18, 1997.

[9] M.G. Gray and R.M. Roberts. Object-based programming in Fortran 90. *Computers in Physics*, 11:355, 1997.

[10] V.K. Decyk, C.D. Norton, and B.K. Szymanski. How to support inheritance and run-time polymorphism in Fortran 90. *Computer Physics Communications*, 115:9–17, 1998.

[11] V.K. Decyk, C.D. Norton, and B.K. Szymanski. How to express C++ concepts in Fortran 90. *Sci. Program.*, 6(4):363–390, 2000.

[12] V.K. Decyk and C.D. Norton. A simplified method for implementing run-time polymorphism in Fortran95. *Sci. Program.*, 12(1):45–55, 2004.

[13] C.D. Norton, V.K. Decyk, B.K. Szymanski, and H.Gardner. The transition and adoption to modern programming concepts for scientific computing in Fortran. *Scientific Programming*, 15(1):27–44, 2007.

[14] C.D. Norton, B.K. Szymanski, and V.K. Decyk. Object-oriented parallel computation for plasma simulation. *Communications of the ACM*, 38(10):88–100, 1995.

[15] L. Machiels and M.O. Deville. Fortran 90: an entry to object-oriented programming for the solution of partial differential equations. *ACM Transactions on Mathematical Software*, 23(1):32, 1997.

[16] J. Qiang, R.D. Ryne, and S. Habib. Parallel object-oriented design in Fortran for beam dynamics simulations. In *Proceedings of the 1999 Particle Accelerator Conference (Cat No 99CH36366) PAC-99*, volume 1, pages 366–368, 1999.

[17] J. Qiang, R. Ryne, and S. Habib. Implementation of object-oriented design with Fortran language in beam dynamics studies. In Kwok Ko and Robert Ryne, editors, *Proceedings of the 1998 International Computational Accelerator Physics Conference (ICAP '98)*, page 87, 2001.

[18] J.E. Akin and M. Singh. Object-oriented Fortran 90 p-adaptive finite element method. *Advances in Engineering Software*, 33(7-10):461, 2002.

[19] D. McCormack. Generic programming in Fortran with Forpedo. *ACM Fortran Forum*, 24(2):18–29, August 2005.

[20] M.L. Hall. The CÆSAR code package. Technical Report LA-UR-00-5568, May 2007.

[21] M. Metcalf, J. Reid, and M. Cohen. *Fortran 95/2003 Explained*. Oxford University Press, 2004.

[22] T.R. Hopkins. Restructuring software: A case study. *Softw. Pract. Exper.*, 26(8):967–982, 1996.

[23] T. Hopkins. Renovating the collected algorithms from acm. *ACM Transactions on Mathematical Software*, 28(1):59, 2002.

[24] C. Greenough and D.J. Worth. The transformation of legacy software: Some tools and a process (version 3). Technical Report TR-2004-012, 2006.

[25] L. Hatton. Does OO sync with how we think? *IEEE Software*, 15(3):46–54, 1998.

[26] D. Joiner. Encapsulation, inheritance and the platypus effect. http://www.advogato.org/article/83.html, 2000.

A Example of Run-time Polymorphism

Source code for `Shape` base class and its subclasses, `Circle` and `Square`.

```
! The Shape_class module. Defines a Shape type and methods

module Shape_class

    implicit none

    ! Define Shape type
    type Shape
        private
        character*12 :: name
    end type ! Shape

    interface print
        module procedure print_Shape
    end interface

    contains

    subroutine setName(self, theName)
        type(Shape), intent(inout) :: self
        character*12 :: theName
        self%name = theName
    end subroutine ! setName

    function getName(self)
        character*12 :: getName
        type(Shape), intent(in) :: self
        getName = self%name
    end function ! getName

    subroutine print_Shape(self)
        type(Shape), intent(in) :: self
        print *, "I am a ",self%Name
    end subroutine ! print_Shape

end module ! Shape_class

! The Circle_class module. Defines a circular shape.
module Circle_class

    ! Bring Shape_class into scope
    use Shape_class

    implicit none

    ! Define Circle type
    type Circle
```

```
    private
    type(Shape) :: super          ! Shape superclass
    real :: centre_x, centre_y    ! centre coordinates
    real :: radius                ! radius
end type ! Circle

interface new
    module procedure init_Circle
end interface

interface print
    module procedure print_Circle
end interface

interface move
    module procedure move_Circle
end interface

contains

subroutine init_Circle(self, x,y, r)
    type(Circle), intent(out) :: self
    real, intent(in) :: x,y               ! centre coordinates
    real, intent(in) :: r                 ! radius

    call setName(self%super, "Circle      ")
    self%centre_x = x
    self%centre_y = y
    self%radius = r
end subroutine ! init_Circle

subroutine print_Circle(self)
    type(Circle), intent(in) :: self

    call print_Shape(self%super)
    print *, "Centre - (", self%centre_x, ",", self%centre_y, "), ", &
             "Radius - ", self%radius

end subroutine ! print_Circle

! Move the circle by vector (x,y)
subroutine move_Circle(self, x,y)
    type(Circle), intent(inout) :: self
    real, intent(in) :: x, y

    self%centre_x = self%centre_x + x
    self%centre_y = self%centre_y + y

end subroutine ! move_Circle
```

```
end module ! Circle_class

! The Square_class module. Defines a square shape.

module Square_class

    ! Bring Shape_class into scope
    use Shape_class

    implicit none

    ! Define Square type
    type Square
        private
        type(Shape) :: super          ! Shape superclass
        real :: centre_x, centre_y    ! centre coordinates
        real :: side                  ! length of side
    end type ! Square

    interface new
        module procedure init_Square
    end interface

    interface print
        module procedure print_Square
    end interface

    interface move
        module procedure move_Square
    end interface

    contains

    subroutine init_Square(self, x,y, l)
        type(Square), intent(out) :: self
        real, intent(in) :: x,y               ! centre coordinates
        real, intent(in) :: l                 ! length of side

        call setName(self%super, "Square      ")
        self%centre_x = x
        self%centre_y = y
        self%side = l
    end subroutine ! init_Square

    subroutine print_Square(self)
        type(Square), intent(in) :: self

        real :: x,y,side
        x = self%centre_x
        y = self%centre_y
```

```
        side = self%side

        call print_Shape(self%super)
        print *, "Coordinates - (", x-side/2.0, ",", y-side/2.0, ") ", &
                                "(", x+side/2.0, ",", y-side/2.0, ") ", &
                                "(", x-side/2.0, ",", y+side/2.0, ") ", &
                                "(", x+side/2.0, ",", y+side/2.0, ")"

    end subroutine ! print_Square

    ! Move the square by vector (x,y)
    subroutine move_Square(self, x,y)
        type(Square), intent(inout) :: self
        real, intent(in) :: x, y

        self%centre_x = self%centre_x + x
        self%centre_y = self%centre_y + y

    end subroutine ! move_Square

end module ! Square_class
```

We define the polymorphic object using pointers to the circle and square objects.

```
! Module to define the polymorphic behaviour of subclasses of shape.

module poly_Shape_class

    ! Bring shape subclasses into scope
    use Circle_class
    use Square_class

    ! The routines to implement run-time polymorphism are private as they are not designed
    ! to be called directly but through interfaces. See below.
    private :: poly_init, assign_circle, assign_square
    private :: poly_print

    ! Define poly_Shape type
    type poly_Shape
        private
        type(Circle), pointer :: pCircle
        type(Square), pointer :: pSquare
    end type ! poly_Shape

    interface new
        module procedure poly_init
    end interface

    interface poly
        module procedure assign_circle, assign_square
    end interface
```

```
interface print
    module procedure poly_print
end interface

interface move
    module procedure poly_move
end interface

contains

! Initialise poly_Shape with null pointers
subroutine poly_init(self)
    type(poly_Shape), intent(out) :: self

    nullify(self%pCircle)
    nullify(self%pSquare)
end subroutine ! poly_init

! Assign circle to poly_Shape
function assign_circle(theCircle) result(shape)
    type(poly_Shape) :: shape
    type(Circle), target, intent(in) :: theCircle

    shape%pCircle => theCircle
    nullify(shape%pSquare)
end function ! assign_circle

! Assign square to poly_Shape
function assign_square(theSquare) result(shape)
    type(poly_Shape) :: shape
    type(Square), target, intent(in) :: theSquare

    shape%pSquare => theSquare
    nullify(shape%pcircle)
end function ! assign_square

! Print a shape. This is where the polymorphism is illustrated.
subroutine poly_print(self)
    type(poly_Shape), intent(in) :: self

    if (associated(self%pCircle)) then
        call print(self%pCircle)
    else if (associated(self%pSquare)) then
        call print(self%pSquare)
    end if

end subroutine ! poly_print

! Move a shape. Make sure we can change data polymorphically.
```

```
    subroutine poly_move(self, x,y)
        type(poly_Shape), intent(inout) :: self
        real, intent(in) :: x,y

        if (associated(self%pCircle)) then
            call move(self%pCircle, x,y)
        else if (associated(self%pSquare)) then
            call move(self%pSquare, x,y)
        end if

    end subroutine ! poly_print

end module ! poly_Shape_class
```

We can use this object in code as follows.

```
! Main program to test run-time polymorphism of shape library

program Main

    ! Bring poly_Shape_class into scope (we also get Circle_class and Square_class)
    use poly_Shape_class

    implicit none

    type(Square) square1
    type(Circle) circle1
    type(poly_Shape) shape1

    call new(square1, 0.0, 0.0, 1.0)
    call new(circle1, 0.0, 0.0, 1.0)
    call new(shape1)

    call print(square1)

    ! Create a polymorphic shape for the square
    shape1 = poly(square1)

    print *, 'Move the square (-2,1.3)'
    call move(square1, -2.0, 1.3)
    call printShape(shape1)

    call print(circle1)

    ! Create a polymorphic shape for the circle
    shape1 = poly(circle1)

    print *, 'Move the circle (2.22,-1.21)'
    call move(circle1, 2.22, -1.21)
    call printShape(shape1)
```

```
end program ! Main

! Subroutine to print a shape
subroutine printShape(theShape)
    ! Bring poly_Shape_class into scope
    use poly_Shape_class

    type(poly_Shape), intent(in) :: theShape

    call print(theShape)

end subroutine ! printShape

! Subroutine to move a shape by vector (x,y)
subroutine moveShape(theShape, x,y)
    ! Bring poly_Shape_class into scope
    use poly_Shape_class

    type(poly_Shape), intent(inout) :: theShape
    real, intent(in) :: x, y

    call move(theShape, x,y)

end subroutine ! printShape
```

B Example of Templates with Forpedo

The source code containing the template definitions.

```
#definetype WorldIdType Int  integer
#definetype WorldIdType Real real

module HelloWorld<WorldIdType>

  @WorldIdType :: worldId<WorldIdType>

contains

  subroutine setId<WorldIdType>(id)
    @WorldIdType :: id
    worldId<WorldIdType> = id
  end subroutine

  subroutine print<WorldIdType>()
    print *,'Hello World "',
      worldId<WorldIdType>,'"'
  end subroutine

end module
```

The code produced by Forpedo

```
module HelloWorldInt

  integer :: worldIdInt

contains

  subroutine setIdInt(id)
    integer :: id
    worldIdInt = id
  end subroutine

  subroutine printInt()
    print *,'Hello World "',worldIdInt,'"'
  end subroutine

end module

module HelloWorldReal

  real :: worldIdReal

contains

  subroutine setIdReal(id)
    real :: id
    worldIdReal = id
  end subroutine

  subroutine printReal()
    print *,'Hello World "',worldIdReal,'"'
  end subroutine

end module
```

C Example of Run-time Polymorphism with Forpedo

Define module `AnimalMod` containing an animal age in years and modules `DogMod` and `CatMod` that use `AnimalMod` and define the `Dog` and `Cat` type as follows.

```
module AnimalMod

  type Animal
    integer :: ageInYears
  end type

end module
```

```
module DogMod
  use AnimalMod

  type Dog
    type (Animal) :: super
  end type

  interface makeSound
    module procedure makeSoundDog
  end interface

  interface increaseAgeInAnimalYears
    module procedure increaseAgeInAnimalYearsDog
  end interface

  interface increaseAgeAndGetAnimalYears
    module procedure increaseAgeAndGetAnimalYearsDog
  end interface

contains

  subroutine makeSoundDog(self)
    type (Dog) :: self
    print *,'Woof!'
  end subroutine

  subroutine increaseAgeInAnimalYearsDog(self, increase)
    type (Dog)             :: self
    integer, intent(in)    :: increase
    self%super%ageInYears = self%super%ageInYears + increase * 7  ! Dog year is 7 human years
  end subroutine

  function increaseAgeAndGetAnimalYearsDog(self, increase) result(returnValue)
    type (Dog)          :: self
    integer, intent(in) :: increase
    integer             :: returnValue
    call increaseAgeInAnimalYearsDog(self, increase)
    returnValue = self%super%ageInYears
  end function

end module


module CatMod
  use AnimalMod

  type Cat
    type (Animal) :: super
  end type
```

```
  interface makeSound
    module procedure makeSoundCat
  end interface

  interface increaseAgeInAnimalYears
    module procedure increaseAgeInAnimalYearsCat
  end interface

  interface increaseAgeAndGetAnimalYears
    module procedure increaseAgeAndGetAnimalYearsCat
  end interface

contains

  subroutine makeSoundCat(self)
    type (Cat) :: self
    print *,'Meeow!'
  end subroutine

  subroutine increaseAgeInAnimalYearsCat(self, increase)
    type (Cat)          :: self
    integer, intent(in) :: increase
    self%super%ageInYears = self%super%ageInYears + increase * 6  ! Cat year is 6 (?) human years
  end subroutine

  function increaseAgeAndGetAnimalYearsCat(self, increase) result(returnValue)
    type (Cat)          :: self
    integer, intent(in) :: increase
    integer             :: returnValue
    call increaseAgeInAnimalYearsCat(self, increase)
    returnValue = self%super%ageInYears
  end function

end module
```

Now we can define the module that implements run-time polymorphism via a Forpedo protocol.

```
#protocol AnimalProtocol AnimalProtocolMod ! We define type AnimalProtocol in
                                           ! module AnimalProtocolMod

#method makeSound                         ! There will be a subroutine makeSound in
type(AnimalProtocol), intent(in) :: self ! AnimalProtocolMod with the 'self' argument
#endmethod

#method increaseAgeInAnimalYears increase       ! This subroutine will have 2 args -
type(AnimalProtocol), intent(inout) :: self     ! self and increase
integer, intent(in)                 :: increase
#endmethod
```

```
#funcmethod increaseAgeAndGetAnimalYears increase,returnValue
type(AnimalProtocol), intent(inout) :: self        ! This function will have 2 args as above
integer, intent(in)                 :: increase    ! and a return value
integer                             :: returnValue
#endmethod

! These are the types and their modules that are polymorphic
#conformingtype Dog DogMod
#conformingtype Cat CatMod

#endprotocol
```

Then we can create a main program that uses the `Dog` and `Cat` types and the Forpedo created `AnimalProtocol` type.

```
program Main
  use AnimalProtocolMod
  use DogMod
  use CatMod
  type (Dog), pointer   :: d
  type (Cat), pointer   :: c
  type (AnimalProtocol) :: p

  allocate(d,c)

  ! Initialization. Should use constructor, but left out for demo purposes.
  d%super%ageInYears = 1
  c%super%ageInYears = 1

  ! Assign polymorphic 'pointer' to Dog
  p = d

  ! Pass pointer to a subroutine that knows nothing about the concrete type Dog
  call doStuffWithAnimal(p)
  print *,"Dog's age is now ",d%super%ageInYears

  ! Repeat for Cat. Results will be different, though subroutine call is the same.
  p = c
  call doStuffWithAnimal(p)
  print *,"Cat's age is now ",c%super%ageInYears

  ! Try again with a function, rather than a subroutine
  print *,"Cat's age is now ", increaseAgeAndGetAnimalYears(p,1)

  deallocate(c,d)

contains

  subroutine doStuffWithAnimal(a)
    type (AnimalProtocol) :: a
    call makeSound(a)
```

```
    call increaseAgeInAnimalYears(a, 2)
  end subroutine

end program
```

Forpedo creates the `AnimalProtocolMod` module listed below.

```
!-----------------------------------------------------------------------------------------------
! This protocol module was generated by Forpedo (http://www.macanics.net/forpedo).
! Do not edit this module directly. Instead, locate the forpedo input file used to generate
! it, and make changes there. When you are ready, regenerate this file with Forpedo.
!-----------------------------------------------------------------------------------------------
module AnimalProtocolMod
  use DogMod
  use CatMod
  implicit none

  integer, parameter, private :: DogId = 0
  integer, parameter, private :: CatId = 1

  type AnimalProtocol
    private
    integer :: concreteTypeId = -1
    type (Dog), pointer :: DogPtr
    type (Cat), pointer :: CatPtr
  end type

  interface assignment(=)
    module procedure assignToType0
    module procedure assignToType1
  end interface

  interface makeSound
    module procedure makeSoundProt
  end interface
  interface increaseAgeInAnimalYears
    module procedure increaseAgeInAnimalYearsProt
  end interface
  interface increaseAgeAndGetAnimalYears
    module procedure increaseAgeAndGetAnimalYearsProt
  end interface


  private :: assignToType0
  private :: assignToType1
  private :: makeSoundProt
  private :: increaseAgeInAnimalYearsProt
  private :: increaseAgeAndGetAnimalYearsProt

contains
```

```
subroutine assignToType0(self,concreteType)
  type (AnimalProtocol), intent(out) :: self
  type (Dog), intent(in), target :: concreteType
  self%DogPtr => concreteType
  self%concreteTypeId = DogId
end subroutine

subroutine assignToType1(self,concreteType)
  type (AnimalProtocol), intent(out) :: self
  type (Cat), intent(in), target :: concreteType
  self%CatPtr => concreteType
  self%concreteTypeId = CatId
end subroutine


subroutine makeSoundProt(self)
  type(AnimalProtocol), intent(in) :: self
  select case (self%concreteTypeId)
  case (DogId)
    call makeSound(self%DogPtr)
  case (CatId)
    call makeSound(self%CatPtr)
  case default
    print *,"Invalid case in makeSoundProt"
    stop
  end select
end subroutine

subroutine increaseAgeInAnimalYearsProt(self, increase)
  type(AnimalProtocol), intent(inout) :: self
  integer, intent(in)                 :: increase
  select case (self%concreteTypeId)
  case (DogId)
    call increaseAgeInAnimalYears(self%DogPtr, increase)
  case (CatId)
    call increaseAgeInAnimalYears(self%CatPtr, increase)
  case default
    print *,"Invalid case in increaseAgeInAnimalYearsProt"
    stop
  end select
end subroutine

function increaseAgeAndGetAnimalYearsProt(self, increase) result(returnValue)
  type(AnimalProtocol), intent(inout) :: self
  integer, intent(in)                 :: increase
  integer                             :: returnValue
  select case (self%concreteTypeId)
  case (DogId)
    returnValue = increaseAgeAndGetAnimalYears(self%DogPtr, increase)
  case (CatId)
```

```
      returnValue = increaseAgeAndGetAnimalYears(self%CatPtr, increase)
    case default
      print *,"Invalid case in increaseAgeAndGetAnimalYearsProt"
      stop
    end select
  end function

end module
```

Index