A FORTRAN preprocessor to support encapsulated data abstraction definitions

W. Burton
School of Computing Studies, University of East Anglia, Norwich NR4 7TJ

In an encapsulated data abstraction definition a data type and the set of all operations which may access or modify instances of the type are defined together.

A simple extension to FORTRAN and a preprocessor to translate from the extended FORTRAN to standard FORTRAN are described. The extended FORTRAN supports encapsulated data abstraction definitions and permits the writing of clear and reliable programs. The preprocessor can do extensive error checking and produce efficient FORTRAN programs as its output.

Definitions of stacks and dynamic arrays are given as examples.

(Received May 1978)

1. Introduction

The proper use of subprograms leads to control oriented modularity and facilitates the top down design of large programs. The correctness of individual modules, or subprograms, may be verified independently. Libraries of commonly used functions and subroutines may be maintained to speed the development of new programs. Two or more modules with identical specifications may be used interchangeably, so that several modules may be tested and the one giving the best performance in a given environment may be used. The modularity resulting from the use of subroutines and functions can greatly improve the readability of a program and make both modification and error detection easier by localising the code requiring a change or correction.

In recent years, data oriented modularity has been widely advocated (Dahl, Dijkstra and Hoare, 1972; Hoare, 1972; Liskov and Zilles, 1974; Tennent, 1977). Simula 67 (Dahl, Myhrhaug and Nygaard, 1970) supports data oriented modules called classes. A number of more recent languages provide similar features (Geschke, Morris and Satterthwaite, 1977; Liskov et al., 1977; Wulf, London and Shaw, 1976).

Classes provide essentially the same advantages as those listed above for the control oriented subroutine and function modules. A class is basically a data type together with a list of the permitted operations which may be performed on instances (or items) of the type. For example, a stack may be a linked list together with push, pop and perhaps one or two other operations.

When a programmer uses a library SIN function he must know what the effect of the SIN function is but does not need to know how the function is computed. Similarly, to use a stack, a programmer must understand the effect of push and pop but does not need to know how the data structure is represented. If a better stack representation or SIN algorithm is implemented, the original source program should be able to use the more efficient implementation without change.

There may be a number of instances of a given class. For example, a programmer might use a number of stacks, just as he might use a number of integer variables or might invoke the SIN function a number of times.

A simple extension to FORTRAN which allows classes to be defined and used is described in Section 2. A simple preprocessor which can translate the extended FORTRAN, which we shall call Classy FORTRAN, to standard FORTRAN is described in Section 3. The syntax of Classy FORTRAN makes possible a number of preprocess time and optional run time checks. Any reader who has tried to debug a standard FORTRAN program which uses complicated data structures will appreciate the discussion on error detection in Section 4. In Section 5 we will see that efficient programs may be produced by the preprocessor if certain subroutine calls are replaced with inline code substitution. Some final remarks are offered in Section 6.

2. Classy FORTRAN

We will describe Classy FORTRAN through a series of examples. The ideas are simple and the implementation of a preprocessor is straightforward.

Fig. 1 shows a main program which uses two stacks. The definition of a class STACK in a separate module is assumed. The variables A, B and C are all references (pointers) to stacks. The first two assignment statements cause two stacks (or instances of class STACK) to be created. Storage is dynamically allocated and initialisation is carried out as

| LOGICAL TEST |
| REF(STACK) A,B,C |
| INTEGER 1,J,K,L |
| A = NEW(STACK) |
| B = NEW(STACK) |
| A.PUSH(1) |
| A.PUSH(2) |
| A.PUSH(11) |
| A.PUSH(12) |
| A.PUSH(13) |
| C = A |
| C.POP(1) |
| C = B |
| C.POP(J) |
| B.POP(K) |
| C.POP(L) |
| B.EMPTY(TEST) |
| IF(TEST) CALL FREE(B) |
| C = .NIL. |
| WRITE(6,10) I,J,K,L |
| 10 FORMAT(1H 4,110) |
| STOP |
| END |

Fig. 1 Use of a STACK
specified in the definition of CLASS STACK. A STACK is initially empty. Two values are pushed on to stack A and three on to stack B. The assignment \( C = A \) causes C to refer to the same stack as \( A \). The stack is not copied. Therefore the first POP assigns 2 to I and leaves only a single item on the stack referred to by both \( A \) and \( C \). The EMPTY operation for a stack sets a logical argument. Since \( B \) is empty in the example, the IF statement causes the STACK to which \( B \) refers to be returned to free storage and \( B \) to be implicitly assigned a .NIL. value. FREE is a generic function which returns the storage occupied by a class instances to the run time storage allocation package. The argument is always reset to .NIL. At this point \( C \) is a dangling reference which is explicitly reset to .NIL. The WRITE statement outputs:

| 2 | 13 | 12 | 11 |

A variable may be declared to be of type REF(STACK) but not of type STACK. If \( A \) is a variable of type REF(STACK) then \( A \) may only be used in the following contexts:

1. If OP is an operation (which may or may not have parameters) defined for items of class STACK then the statement

\[
A.\text{OP}
\]

is permitted.

2. A new STACK may be created and initialised, with \( A \) being set to refer to the new STACK, by the statement

\[
A = NEW(\text{STACK})
\]

where NEW is a key word. NEW is not a reserved word. STACK cannot be both a class name and a variable name, so the meaning of NEW is clear from context.

3. If \( B \) is a variable of type REF(STACK) then

\[
A = B \text{ and } B = A
\]

are both permitted.

4. \( A \) may be used as an actual (dummy) argument in a function or subroutine call (statement) provided the corresponding dummy (actual) argument is also of type REF(STACK).

The value .NIL. may also be used as an actual argument if the corresponding dummy argument is of type REF(STACK).

5. \( A \) may be assigned the value .NIL. .

6. If \( B \) is either a variable of type REF(STACK) or .NIL. then \( A \) and \( B \) may be compared by the operators .EQ. and .NE. . Equality occurs if and only if the same instance is referenced by both values.

7. The statement

\[
\text{CALL FREE}(A)
\]

is permitted. If the representation of \( A \) contains references to areas of memory which are not otherwise reachable, then these areas will become totally unreachable without being returned to free storage.

Arrays of reference variables are allowed. Reference variables may not occur in COMMON or EQUIVALENCE statements. Fig. 2 contains the definition of a STACK implemented as a linked list. In a top down manner CLASS STACK uses CELLS as defined in Fig. 3.

A class definition has the following form:

```plaintext
<class statement>
<argument specifications>
<component list>
<operation list>
<initialisation part>
END <class name>
```

The class statement and argument specifications give the name of the class and a description of the arguments used in defining a new instance of the class. The argument specifications are optional in classes, as with functions and subroutines. However, argument specifications, if given, may not be mixed with local variable or component declarations.

For each instance of a particular class there is a separate copy of each component in the component list. For example, each instance of class CELL has its own \( I \) and \( NEXT \) values. These values exist from the time the CELL is created until it is returned to free storage by a call to FREE. Therefore, at any given time there may be a number of different \( I \) and \( NEXT \) values.

Operations have much the same form as functions and subroutines except that variables in the component list may be accessed. It is often useful to think of each instance of a class having its own copy of each operation since the components manipulated by a given operation will be different for different instances of the class.

The initialisation part is executed for each instance of a class at the time that instance is created. For example, when a STACK is created, the pointed TOP is set to .NIL. . When a CELL is created, copies of the arguments are placed in the CELL components \( I \) and \( NEXT \). The <initialisation part> may include local variable declarations.

Components may be of type INTEGER, REAL or LOGICAL or be arrays of integers, reals or logicals. Arrays may be of variable size. The same rules which govern the specification of arrays in subprograms govern the declaration of arrays in class instances. For example, dynamic arrays can be defined using CLASS ARRAY shown in Fig. 4.

The components of a class instance are global to the class definition. Dummy arguments in the CLASS statement are local to the initialisation part and dummy arguments in oper-
procedures are local to the particular operation. Both operations and the initialisation part may include the declaration of local variables. All local variables and class components used within a class definition must be explicitly declared. Otherwise there could be confusion as to whether an identifier represented a local variable or a global class component.

A class instance may be accessed or modified via its operations. The class components are not directly accessible outside the class definition. This makes it impossible for a user to accidentally (or otherwise) create an ill-formed STACK or assign to pointers the locations of individual CELLS in the STACK. Only PUSH, POP and EMPTY may be used to access or modify a STACK.

The keyword OPERATION may be preceded by the keyword PRIVATE. A private operation can only be used by the initialisation part and other operations within the class definition. This makes greater security possible when misuse of an operation could cause problems. For an example of a private operation (using different syntax) see Wegbreit and Spitz (1976).

The keywords OPERATION, PRIVATE and CLASS may all be preceded by the keyword INLINE. Use of INLINE does not affect the meaning of a class definition. The effect of the INLINE qualification on efficiency is discussed in Section 5. An inline class or operation must be defined textually before its first use.

There are a few restrictions in Classy FORTRAN to avoid name clashes. The names FREE, CFNEW, CFERR, and MEMORY are reserved for program units in the run time package. In addition, the names INDEX, ISTORE, RSTORE and LSTORE should not be used within any class definition. Finally, each user defined name must differ from each class name within the first three characters, the names of any two

classes must differ within the first three characters, the names of any two operations in a given class must differ within the first three characters, and the fourth through sixth characters of a class name must differ from the first three characters of the name of each operation in the class. Each class name must be at least three characters long.

3. Implementation

We will first consider the problem of translating a program which uses classes into standard FORTRAN, and then go on to consider the translation of class definitions.

The Classy FORTRAN program shown in Fig. 1 translates to the standard FORTRAN program shown in Fig. 5. Notice that each subroutine name is generated from the first three characters of the class name and the first three characters of the operation name, with a blank inserted for readability. The translation is quite straightforward.

Some error checking should be done at translation time. At the very least, it should be established that a variable of type REF(<class>) is never assigned a value other than .NIL., NEW(<class> <parameters>) or the value of some other variable of the same type. A more complete check on the contexts in which a class instance is used would be desirable.

The preprocessor produces a separate subroutine for each operation and initialisation part. The creation of a new class instance by an initialisation part clearly necessitates dynamic storage allocation. The call

CALL CFNEW(INDEX,LENGTH)

will cause a block of storage, of length LENGTH, to be allocated from a common block, using the modified first-fit method of allocation (Knuth, 1968). Upon return, INDEX will index the last storage unit in the block. The preprocessor
must establish a correspondence between storage units in a block of allocated storage and the components of a class instance. Fig. 6(a), (b) and (c) show the result of preprocessing the class definition in Fig. 3.

To illustrate how dynamic arrays are handled, Fig. 7 gives the subroutine to create a new ARRAY as defined in Fig. 4. Notice that a hidden component gives the size of VALUE. This hidden component is used in the operations for indexing and optional error checking. The behaviour of the preprocessor should be clear from these examples.

4. Error checking
The syntax of class definitions and usage makes a high degree of error checking possible.

As mentioned earlier, it is desirable to check at compile time to make sure that no reference type variable is used in an illegal context. A number of further error checks may be made if run time overheads are accepted and separate preprocessing of modules is not required.

Run time checks, which can be turned on or off at the time of preprocessing, include subscript checking for arrays which are components in class instances, and pointer validity checks.

With full error checking, all reference variables are initialised to the illegal subscript value, -1, in DATA statements. If FORTRAN subscript checking is turned on, any attempt to refer to a component of an item identified by a .NIL or uninitialised pointer will lead to an invalid subscript in a reference to ISTORE.

Dangling references are harder to catch. Each block of storage is allocated a serial number (in a cyclic fashion if necessary). Reference variables contain the serial number and index packed together. The same serial number is packed in the header of the storage block. Whenever an operation is first
SUBROUTINE ARRAY(INDEX, M)
C *WORKSPACE DECLARATION
   REAL RSTORE(10000)
   LOGICAL LSTORE(10000)
   COMMON /MEMORY/ISTORE(10000)
   EQUIVALENCE (RSTORE(1), LSTORE(1), ISTORE(1))
C *STORAGE ASSIGNMENTS
C *VALUE: RSTORE(INDEX-SIZE(VALUE)) THROUGH RSTORE(INDEX-1)
C *SIZE(VALUE): ISTORE(INDEX)
C *EXECUTABLE CODE
   CALL CPNEW(INDEX, M+1)
   ISTORE(INDEX)=M
   RETURN
END

Fig. 7 Creation and initialisation of a new ARRAY

INTEGER INDEX
   .
   .
   .
CALL CPNEW(INDEX, 2)
ISTORE(INDEX-1)=13
ISTORE(INDEX)=ISTORE(B)
ISTORE(B)=INDEX

Fig. 8(a) INLINE expansion of B.PUSH(13)

INTEGER TEMP
   .
   .
   .
L=ISTORE(ISTORE(C)-1)
TEMP=ISTORE(ISTORE(C))
CALL FREE(ISTORE(C))
ISTORE(C)=TEMP

Fig. 8(b) INLINE expansion of C.POP(L)

entered, the current validity of the pointer is established and a simple index is stripped off. While this method is not absolutely foolproof (for example in cases where serial numbers are eventually repeated, or a data word is mistaken as a header word), it is extremely unlikely that any given dangling reference will go undetected during program debugging. Of course, run time overheads for this type of checking are extremely high. It is essential that such checking be made optional, primarily for use in program development.

If separate preprocessing of program modules is not desired, then some additional error checking is possible. To avoid reference variables being assigned invalid values, it is important to check that actual and dummy arguments agree in type. Operation name checking is also of value. Suppose there happens to be a subroutine STATOP and a user tries to use the operation TOP with a STACK. (Recall that no such operation is defined for STACKs). Separate preprocessing would translate references to operation TOP into calls to subroutine STATOP.

Finally, the preprocessor could ensure that PRIVATE operations are not used outside the class definitions in which they occur. These simple checks will catch a large number of common errors.

5. Efficiency
As the reader has no doubt observed, the use of classes tends to result in the invocation of a large number of very simple operations. If these operations are implemented as subroutines, the resulting programs are likely to be very inefficient. This problem can be solved if certain operations and initialisation parts are substituted inline, with suitable parameter substitutions and systematic name changes as required, at points where these are referenced.

The occurrence of the keyword INLINE in a CLASS or OPERATION statement indicates to the preprocessor that it is desirable that the initialisation part or operation body, respectively, be substituted inline where referenced. This, of course, excludes the possibility of separate preprocessing.

Fig. 8(a) and (b) show the inline expansion of B.PUSH(13) and C.POP(L), respectively, which would result from declaring operations PUSH, POP, VALUE and LINK and class CELL all INLINE. (We are assuming that subscribed subscripts are allowed, as in FORTRAN 77 (ANS, 1976). The resulting code is much the same as would be written by a user working directly in FORTRAN.

A preprocessor to support inline substitution of subroutine bodies would be useful even without the use of data classes. With recent moves toward structured programming and modularity, subroutines are being used for readability, rather than to limit object code size, with increasing regularity. This leads to more subroutines. On the other hand, with virtual storage machines the invocation of a subroutine may result in a page turn, making unnecessary subroutine calls expensive. Requiring that logical and physical modules be the same seems, to this author, to be as silly as requiring logical and physical records to coincide in file processing applications.

6. Conclusion
A simple method for providing data abstractions in FORTRAN and a preprocessor to support the resulting extension to FORTRAN have been described. A data abstraction definition consisting of the definition of a new data type together with all the definitions of permissible operations for the data type can be encapsulated in a single module.

Libraries of data abstraction definitions such as stack, queue, tree, etc. can be provided, thereby greatly simplifying the writing of programs requiring structures of these types.

If the physical representation of a data structure must be changed, the proposed modularity makes the modification trivial. References to the data structure at the source level remain unchanged. Only the encapsulated definition is changed.

The use of data abstractions makes programs much more readable (contrast lines 10 and 16 of Fig. 1 to Fig. 8(a) and (b)). Data abstractions can be designed in a top down manner. For example the STACK definition in Fig 2 uses CELLSs as defined in Fig. 3. If maximum error checking is turned on, the syntax of the class definitions and references permits a high degree of error checking. Any reader who has ever tried to implement directly nontrivial data structures in FORTRAN, using array indices for pointers, will appreciate the value of this.

On the other hand, the preprocessor can generate efficient programs by using inline code substitution, making Classy...
FORTRAN suitable for writing production programs.

We have tried to keep the proposed FORTRAN extension as small as possible, and have excluded from consideration a number of other features which could be useful in data class definitions (Geschke and Mitchell, 1975; Geschke, Morris and Satterthwaite, 1977; Liskov et al., 1977; Shaw, Wulf and London, 1977; Wulf, London and Shaw, 1976). This makes mastery of Classy FORTRAN a quick and easy process. It also keeps the preprocessor fairly simple, making it straightforward to implement in a reliable manner. A user desiring more extensive features should probably consider using a language other than FORTRAN.

References


Book reviews

Communications Architecture for Distributed Systems by R. J. Cypher, 1978; 710 pages. (Addison-Wesley, £15.75)

Although distributed systems are currently a major topic of interest and activity in data processing, there are very few good books on the subject. My immediate reaction on receiving this book by Cypher was one of awe; a 711-page work on network architectures. But on second examination, the title says 'communications architecture' (singular) and is actually about IBM's SNA. This was not a disappointment. IBM did a lot of pioneering in this field and any insight into the design and development of this complex range of hardware and software products must be of value to the industry.

The book is 'official' in the sense that it is published as part of a series sponsored by IBM through an editorial board of staff members. This assures us of a certain level of authority and accuracy. But the 'wants' are not there; do not read the book if you want a balanced view. In spite of that Cypher is still essential reading for specialists in networking and distributed systems.

As one would expect, the book is well structured; starting with a good review of non-architected systems it goes on to deal with the trends in network structures, techniques for improving cost/performance on the communications channels, the influences towards greater distribution and nicely summarises the objectives of effective multiple computer networks. These early chapters lay the foundations for the main sections of the book which are concerned with the concepts of SNA, network services, dataflow and transmission control, operational control and advanced functions such as multi-domain networks. The style is as clear as it can be when encumbered by the unavoidable jargon and initials of such a complex system. However, the liberal use of examples and schematics helps a lot.

What is not clear from the book is the extent to which SNA has changed over the last five years. IBM's early announcements were uncompromising; VTAM would be the only access method, SDLC the only line protocol, all control would be centred on the host and so on. Today there are still three access methods (VTAM, TCAM and XTAM) which can be used in an SNA environment; products as recent as the 8100 have support for the BSC link control and the 8100 also has a networking capability not dependent upon a 370 host system.

This issue of central-host control is a major one and the book only touches on it in a final chapter on interfacing to new data networks. SNA and X25 do not go well together. For example, a schematic of 'one possible arrangement of an SNA path information unit within a packet, within a frame' shows no less than eight headers and trailers.

Of course, X25 does not merely specify an interface but also a transmission technique and that technique conflicts strongly with SNA's centralised polling structure. The discussion of these issues makes fascinating reading.

This is not a management book, but is recommended for communications and distributed systems specialists.

David Hedditch (Otley)


This is a slim volume, containing a mere 119 pages, which surveys progress in this field to date under the three headings of Threats, which indicate the scope of the problems; Countermeasures, which indicate means for safeguarding various parts of a computer system and Implemented Systems, which refer to the software and hardware aspects of machine architecture rather than any physical and administrative security techniques. The major part of the book, some 77 pages, deals with countermeasures. The style of presentation is to identify a number of issues in turn such as safeguarding the hardware, the operating system, the terminals, the communication system, etc. and then to discuss each issue briefly with reference to key papers identified in the Bibliography (185 papers). A large number of technical terms are used which are not fully explained, e.g. segmentation, capability addressing, virtual memory, active page registers, inter and intra domain calls, tagged architecture, descriptors, etc. and for the non expert or the student, except a computer science student, this is going to make progress difficult. The authors make no firm recommendations, accept that no system is inviolable and suggest that the best approach is to make the cost of the protection system match the value of the protected data. They conclude by stating that protection promises to be an active area of research for many years to come. I cannot strongly recommend the book since it lacks real substance of its own. However, its brevity means that it is readable and it provides a good set of pointers to anyone starting to take an interest in this area.

D. B. G. Edwards (Manchester)