

# How to Express C++ Concepts in Fortran90

Viktor K. Decyk

Department of Physics and Astronomy  
University of California, Los Angeles  
Los Angeles, CA 90095-1547

&  
Jet Propulsion Laboratory  
California Institute of Technology  
Pasadena, California 91109-8099

email: decyk@physics.ucla.edu

Charles D. Norton

Jet Propulsion Laboratory  
California Institute of Technology  
Pasadena, California 91109-8099

email: nortonc@olympic.jpl.nasa.gov

Boleslaw K. Szymanski

Department of Computer Science  
and  
Scientific Computation Research Center (SCOREC)  
Rensselaer Polytechnic Institute  
Troy, NY 12180-3590

email: szymansk@cs.rpi.edu

## **Abstract**

This paper summarizes techniques for emulating in Fortran90 the most important object-oriented concepts of C++: classes (including abstract data types, encapsulation and function overloading), inheritance and dynamic dispatching.

## I. Introduction

Object-Oriented Programming (OOP) has proven to be a useful programming paradigm for complex programs, especially those modeling “real world problems.” The scientific community has been slower to adopt this paradigm, but even here OOP is beginning to draw a following and even more curious interest. There are a number of reasons for this reticence in the scientific community. One reason is that many scientists who write modest-sized programs for their own needs, are comfortable using Fortran77 and C, and see no reason to change. Others with more complex programs written in Fortran have a great deal invested in their legacy codes and do not want to switch to a new programming language because of the threat to this investment. Adopting OOP means not only learning a whole programming style, but learning a new and unfamiliar language as well. The dominant OO language in the scientific community, C++, is very complex and requires a substantial investment of time to learn how to use effectively. In using C++, there are also concerns about reported poor performance, lack of language and compiler standardization, and lack of standard class libraries for scientific computing. Although many of these concerns are being addressed by the C++ community, the scientific programmer may not know how to evaluate the current situation. Finally, many people have no clear idea of how their scientific productivity will improve by using OOP.

Fortran90 is a modern programming language with many new features which appear to be useful for OOP. Since it is known that OOP is possible in non-OOP languages [1], we decided to test the capabilities of Fortran90 by translating published examples of C++ code from textbooks and journal articles. We discovered that almost all the features of C++ could either be translated directly or emulated without great effort. The major exception was dynamic binding, emulation of which required more effort. As a result, we feel that it is practical to adopt OOP principles in Fortran90. Since Fortran90 is backward compatible with Fortran77, this gives a migration path for evolving toward a new programming style in an incremental fashion. OOP in both Fortran90 and C++ requires applying a methodology and non-OOP programs are possible in either language. However, OOP in Fortran90 requires more self-discipline.

These techniques are useful even if the ultimate goal is to convert a Fortran legacy code to C++. Experience has shown that it takes a year or more to convert a typical scientific code to C++. This is a long time for a scientist to be non-productive. Converting to an OO-style in Fortran90 is much faster and the code can be in continual productive use in the meanwhile. The final step in moving from Fortran90 to C++ is much easier if the classes and objects are already clearly understood.

This paper is also of interest to those who need to work in an environment where Fortran90 and C++ are both used. Knowing how to translate concepts between the languages has proved useful to us in merging Fortran90-based scientific codes with graphics libraries and utility libraries for adaptive mesh refinement written in C++. If advanced features of Fortran90 are used, it is much more challenging to merge Fortran90 and C++ than it is to merge Fortran77 and C. Understanding which features of C++ are difficult to emulate and which are not is also useful in determining which language should be used for which part of a complex project.

Finally, it has been determined that Fortran2000 will be fully object-oriented. New language features will be added for inheritance and run-time polymorphism, while retaining existing language mechanisms for other OO concepts. The techniques described here can be useful as temporary bridges between the non-OO Fortran77 and the fully-OO Fortran2000.

This paper summarizes the techniques we have developed for implementing C++ concepts in Fortran90. It is assumed that the reader has at least a passing familiarity with the

concepts of C++, but not necessarily with Fortran90. For readers unfamiliar with C++, there are many textbooks available. One that we have found useful is Lippman's [2]. For a more extensive explanation of Fortran90, we recommend the book by Ellis et. al. [3] For those who are not familiar with either C++ or Fortran90, we recommend our earlier introductory article [4]. We shall illustrate many of our ideas by using the extended example of a database application which is described by Henderson and Zorn [5] and is used as a benchmark for object-oriented languages. For pedagogical reasons, we have simplified and slightly modified their original code. The reader may wonder why a non-scientific example was chosen to illustrate programming principles of interest to scientists. First, by picking a standard object-oriented benchmark, experts in OOP would not have difficulty recognizing that OOP was possible in Fortran90. The second reason was to keep the example as simple and general as possible.

For the scientific programmer, the personnel database discussed here serves as a model or paradigm for a collection of more complex objects relevant to some scientific problem. For example, instead of students and teachers, plasma physicists could think of collections of different kinds of particles which require different integration schemes, such as electrons with full dynamics, unmagnetized ions, or gyro-kinetic particles integrated with a drift approximation. If the particles share some common code, along with additional code specific to each particle type, then the personnel database corresponds to managing collections of such particles in a multi-species plasma simulation. For electrical engineers, instead of students and teachers, one could imagine managing collections of adaptive mesh types for electromagnetic calculations, where different mesh types are used in different regions of the calculation. Since we are focussing on programming techniques here, we thought that picking a complex example from plasma physics or electromagnetics would have distracted the reader and made the paper more difficult for those unfamiliar with the specific scientific area.

## II. Classes

The most fundamental concept in C++ which must be modeled is the idea of classes. Classes contain a new data type and the procedures that can be performed by the class. The elements (or components) of the data type are the class data members, and the procedures are the class member functions (or methods). We define a class in Fortran90 as a module which contains exactly one abstract data type definition (called a derived type in Fortran90) and the procedures which work exclusively on that type. In addition, a module can also contain data which corresponds to static class members in C++. As an example, consider the Personnel class from Henderson and Zorn's application. A primitive Personnel class can be defined as follows:

```
module Personnel_class
  type Personnel
    integer :: ssn
    character*12 :: firstname, lastname
  end type Personnel
  contains
    subroutine null_Personnel(this)
! Personnel constructor
    type (Personnel), intent (out) :: this
    this%ssn = 0
    this%firstname = ' '      ! blank name
    this%lastname = ' '
    end subroutine null_Personnel
  end module Personnel_class
```

The class data type contains three data members, an integer identifier called `ssn` and two character variables called `firstname` and `lastname`. In addition, this class contains one procedure, a simple constructor, which initializes the component `ssn` to zero and sets the character variables to blank. Since a module name cannot be the same as the derived type name in Fortran90, by convention we append the string `"_class"` to the type name to obtain the module name. Fortran90 is not case sensitive as is C++, but we will use mixed case for textual clarity. Comments are preceded by the `'!'` character rather than the `'/'` string used in C++. Fortran90 also allows free format, but we will continue to use fixed format here. A simple main program which uses this `Personnel_class` is shown below:

```
program personnel_test
  use Personnel_class
  type (Personnel) :: person
  call null_Personnel(person)
end program personnel_test
```

The `USE` statement is a scope operator which brings the class into the scope of the main program. Forward declarations are not needed for procedures in modules in Fortran90 (they are generated automatically), and there is no notion of file scope. (An `INTERFACE` statement is available to provide the functionality of forward declarations for procedures not in modules.) An object of this class (`person`) is created in two steps. First a variable of this type is declared and created (but not initialized), then a constructor procedure is applied to initialize the object:

```

type (Personnel) :: person
call null_Personnel(person)

```

This is different than in C++, where both creation and initialization can be combined into one statement with the `new` operator. (Fortran90 also requires that all declarations appear before any executable statements.) In the procedure `null_Personnel`, note that a reference to the class object always appears explicitly as an argument in Fortran90, and by convention we make it the first argument and call it 'this.' In C++, a reference to the object is available, but is not explicitly declared. Furthermore, the class data members are accessed as components of the dummy 'this' argument. Fortran90 uses the '%' notation to refer to components of a type where C++ uses the '.' notation to refer to components of a structure. Thus in a Fortran90 procedure one writes:

```

this%ssn = 0

```

whereas in C++, one would merely write:

```

ssn = 0;

```

Fortran90 has two ways to represent strings. The most common way is to use variables of type `CHARACTER`, which are declared as follows:

```

character*12 :: firstname

```

Here the variable `firstname` is declared to be 12 bytes in length. Character variables are actually encapsulated objects which know their own length. They are not null-terminated as in C++. The length cannot be set dynamically, but must be known at compile time. A substantial number of intrinsic operators are available for character manipulations in Fortran90.

Let us now enhance this primitive `Personnel` class so that it actually does something useful. We begin by creating a new constructor `init_Personnel` to initialize the object with actual information, as follows:

```

subroutine init_Personnel(this,s,fn,ln)
type (Personnel), intent (out) :: this
integer, intent (in) :: s
character*(*), intent (in) :: fn, ln
this%ssn = s
this%firstname = fn
this%lastname = ln
end subroutine init_Personnel

```

Later we will show how to overload functions so that both constructors have a common name. In this new constructor, the `INTENT(IN)` attribute on the argument `s` means that it will not be modified, and therefore corresponds to the `const` keyword used in dummy arguments in C++. There appears to be no counterpart in C++ to the `INTENT(OUT)` attribute except for the return value of a function. When passed as a dummy argument to a procedure, the length of a character variable does not have to be declared (but can be determined with the `LEN` intrinsic, similar to the `strlen` library function in C++). Now, one can initialize `person` as follows:

```

type (Personnel) :: person

```

```
call init_Personnel(person,1,'PAUL','JONES')
```

Fortran character variables function as a built-in string class and are widely used for this purpose. They have a disadvantage, however, in that the strings are always of fixed length, so that memory can be wasted. As an alternative, it is possible to construct C-style strings as an allocatable array of characters, such as:

```
character*1, dimension(:), allocatable :: firstname
```

The memory for such an array can be dynamically allocated using the ALLOCATE statement (rather than the new statement used in C++), as follows:

```
allocate(firstname(len('PAUL')))
```

where we have allocated an array of 1 byte characters equal in size to the length of the string 'PAUL'. (Arrays of n byte characters can also be allocated in a similar way.) The memory for such arrays is freed with the DEALLOCATE statement (rather than delete), as follows:

```
deallocate(firstname)
```

C-style strings are not commonly used in Fortran90 because not all of the Fortran string manipulation intrinsics are available. For example, string assignment with C-style strings must be done using array constructors:

```
firstname = (/ 'P', 'A', 'U', 'L' /)
```

instead of what one would normally do with character variables:

```
firstname = 'PAUL'
```

Since Henderson and Zorn use C-style strings in their example, however, we will do so here as well. In order to make string assignment simpler, we will write a copy procedure (which we shall call `strcpy`) which will convert a character variable of fixed size to a C-style array of 1 byte characters:

```

subroutine strcpy(s,c)
character, dimension (:), intent (out) :: s
character*(*), intent (in) :: c
do i = 1, max(size(s),len(c))
    s(i) = c(i:i)
enddo
end subroutine strcpy

```

Here we have used the SIZE intrinsic to determine the length of the character array *s* and the LEN intrinsic to determine the length of the character variable *c*.

With this procedure, one can use dynamically allocated arrays of characters while still retaining a simple assignment syntax:

```

call strcpy(firstname,'PAUL')

```

Allocatable arrays cannot be used in derived type definitions (i.e., as class data members), so pointers to arrays must be used instead. Pointers in Fortran90 are objects whose internal state is private. Arrays and pointers to arrays have the same syntax and a pointer to an array can be used whenever an array is expected, including passing it to a procedure. This is quite natural in Fortran90, since arrays are always passed by reference. Thus the expression:

```

ptr(3) = 4

```

will either set the third word of the array *ptr* to 4, if *ptr* is an array, or else it will dereference the corresponding location in memory, if *ptr* is a pointer to an array. In C++ this will also work, but it is common practice to use different syntax for these two cases, where elements of an array are set with the bracket notation:

```

ptr[2] = 4;

```

while the following expression:

```

*(ptr + 2) = 4;

```

will dereference a pointer to an array. Note the difference in subscripting arrays elements in Fortran90 and C++, which is caused by using the Fortran90 default of array referencing from 1 rather than 0. This default can be changed.

Adding a constructor and destructor to our class, we can extend our class definition as follows:

```

    module Personnel_class
! define Personnel type
    type Personnel
        integer :: ssn
        character, dimension(:), pointer :: firstname, lastname
    end type Personnel
    contains
        subroutine init_Personnel(this,s,fn,ln)
! Personnel constructor
        type (Personnel), intent (out) :: this
        integer, intent (in) :: s
        character*(*), intent (in) :: fn, ln
        this%ssn = s
        allocate(this%firstname(len(fn)),this%lastname(len(ln)))
        call strcpy(this%firstname,fn)
        call strcpy(this%lastname,ln)
        end subroutine init_Personnel
!
        subroutine term_Personnel(this)
! Personnel destructor
        type (Personnel), intent (inout) :: this
        deallocate(this%firstname,this%lastname)
        end subroutine term_Personnel
!
        subroutine strcpy(s,c)
        character, dimension (:), intent (out) :: s
        character*(*), intent (in) :: c
        do i = 1, max(size(s),len(c))
            s(i) = c(i:i)
        enddo
        end subroutine strcpy
    end module Personnel_class

```

For convenience, we have put the `strcpy` function in this module. If we were going to do a lot of string manipulations using C-style strings, one would create a string class to do so. To keep the example simple, however, we will not do so here.

One can also overload procedure names for functions in modules with the `INTERFACE` statement. This allows one to use the same name for different procedures. For example, we can overload the name `new` and equate it to the constructor names `null_Personnel` and `init_Personnel`, with the following statements in the module before the `CONTAINS` statement:

```

interface new
    module procedure null_Personnel, init_Personnel
end interface

```

In Fortran90 overloading procedure names requires two steps, whereas in C++ this can be done with one step by merely reusing a procedure name with different argument types. Similarly, one can overload the name `delete` and equate it to `term_Personnel`. As in C++, overloading is possible only if the arguments of the procedure are distinct. In Fortran90, however, the object is an argument and is therefore also taken into account in

resolving overloaded functions, whereas in C++, a different mechanism (polymorphism) is used to overload procedure names which refer to objects in different classes. Constructors and destructors must always be called explicitly in Fortran90, but they are not needed as often as in C++ because Fortran90 always passes arguments by reference, not by value. Destructors are generally needed only if the class contains pointers which must be deallocated. With the current definition of the `Personnel` class, initializing and deleting an object looks like:

```
type (Personnel) :: person
call new(person,1,'PAUL','JONES')
call delete(person)
```

This basic `Personnel` class can be further enhanced with additional features similar to those available in C++. First of all, one can add the `PRIVATE` attribute to the type definition as follows:

```
type Personnel
  private
  integer :: ssn
  character, dimension(:), pointer :: firstname, lastname
end type Personnel
```

This functions just like the `private` keyword in C++ and makes the components of type `Personnel` available only to member functions in the module (class). The default is `PUBLIC`. The `protected` keyword is not available in Fortran90. It is also not possible in Fortran90 to make some data members of `Personnel` `PUBLIC` while keeping others `PRIVATE`.

Procedure names and type definitions in modules can also be made `PUBLIC` or `PRIVATE`. By default, they are `PUBLIC`. For example, the following statement will make the names `init_Personnel` and `term_Personnel` `PRIVATE`.

```
private :: init_Personnel, term_Personnel
```

Fortran has default declarations for variables. One can require all variables to be declared as in C++ with the `IMPLICIT NONE` statement.

Let us add a procedure to this class to print a copy of the `Personnel` record, omitting the identifier `ssn` by default. Such a procedure might look like:

```
subroutine print_Personnel(this,printssn)
type (Personnel), intent (in) :: this
logical, optional, intent (in) :: printssn
if (present(printssn)) then
  if (printssn) write (*,'(i2,a2)',advance='no')
&  this%ssn,': '
endif
Print *, this%firstname, ' ', this%lastname
end subroutine print_Personnel
```

Fortran90 does not have default values for arguments, but `OPTIONAL` arguments can be used for this purpose. The `OPTIONAL` variable `printssn` here is of type `LOGICAL` which is an object whose internal representation is private. There is no automatic conversion between logical types and other types in Fortran90. (In fact, there are no automatic casts

permitted across procedures in Fortran90 at all.) The identifier `ssn` will be printed if the LOGICAL `printssn` is both PRESENT as an actual argument and true. Fortran90 uses the logical operator `.AND.` where C++ uses `&&`. The PRINT \* statement will print on the default output device with default formatting. Note that the PRINT statement is dereferencing the entire arrays pointed to by `this%firstname` and `this%lastname`. The PRINT \* statement will always append a newline character to the output. To suppress this, one must use the `ADVANCE='NO'` specifier in a WRITE statement, which also requires a format specification. This is the opposite situation to C++, where the iostream `cout` requires the manipulator `endl` in order to insert a newline character.

Another useful procedure one can add to this class is extracting the Personnel identifier `ssn`:

```
function getssn_Personnel(this) result(ssn)
type (Personnel), intent (in) :: this
integer :: ssn
ssn = this%ssn
end function getssn_Personnel
```

In Fortran90, the specification RESULT can be used to identify the name of the function result variable, while C++ uses the keyword `return` to identify a result expression. If we overload the name `print` to refer to `print_Personnel`, then the following code extract will initialize a Personnel record and print it along with the identifier:

```
type (Personnel) :: person
call new(person, 1, 'PAUL', 'JONES')
call print(person, .true.)
```

Note that in Fortran90, we invoke a method on an object with the syntax:

```
call print(person, .true.)
```

whereas in C++, the `'.'` syntax would have been used:

```
person.print(1);
```

Another useful addition to this class is the ability to determine how many Personnel records have been created. One way to accomplish this is to add a static class member called `NUM_FILES` to keep track of the number of records. This can be done in Fortran90 by placing the declaration:

```
integer, save :: NUM_FILES = 0
```

anywhere before the CONTAINS statement. Variables inside modules function as global variables for the module. They are in scope whenever the class is in scope (that is, when the module name is declared in a USE statement). Here the SAVE attribute is equivalent to the keyword `static` in C++. Static class members can be initialized inside the class definition, which is not generally allowed in C++. One can keep track of the current number of records by incrementing the global variable `NUM_FILES` inside the procedure `new`, and decrementing it in the procedure `delete`.

There is no notion of a class scope operator in Fortran90. Scope is controlled by the USE statement, and procedure names either must be unique, or overloaded with the

INTERFACE statement. If there is a variable name conflict, one can rename the variable when the Personnel class is “used.” In the following example:

```
use Personnel_class, local_NUM_FILES_name => NUM_FILES
Print *,local_NUM_FILES_name
```

the name local\_NUM\_FILES\_name is now used to refer to the static class member NUM\_FILES. Alternatively, it is possible to make static class members PRIVATE by adding the PRIVATE attribute to the declaration.

```
integer, save, private :: NUM_FILES = 0
```

In that case one must provide a static member function without a ‘this’ reference to read the value of NUM\_FILES, for example:

```
integer function get_num_files()
get_num_files = NUM_FILES
end function get_num_files
```

Here no RESULT specification is used and therefore the function name is used as the result variable.

The final version of the Personnel class is listed in Appendix A. Both the Fortran90 and C++ versions are shown for comparison. The following program will create a record, print out a copy without the identifier, delete it, and finally print out the number of existing records (which should be 0 at this point of execution).

```
program personnel_test
use Personnel_class
type (Personnel) :: person
call new(person,1,'PAUL','JONES')
call print(person)
call delete(person)
Print *,'NUM_FILES=',get_num_files()
end program personnel_test
```

### III. Inheritance

Another important concept in C++ which must be modeled to support object-orientedness is the idea of inheritance. Inheritance allows one to create a hierarchy of classes in which the base class contains the common properties of the hierarchy and the derived classes can modify and specialize these properties. Specifically, a derived class contains all the class data members of the base class and can add new ones. Further, a derived class contains all the class member functions of the base class, and can modify them or add new ones. The value in using inheritance is to avoid duplicating code when creating classes which are similar to one another. As an example, Henderson and Zorn define student records as a type of personnel record through inheritance from personnel. Fortran90 does not directly support this kind of inheritance, but an equivalent relationship can be constructed. Inheritance of class data members is constructed by explicitly including a base class data type in the definition of the derived class data type. For example, if `Student` is derived from `Personnel`, the `Student` type can be expressed as follows:

```
type Student
  type (Personnel) :: personnel
  integer :: nclasses
  character*12, dimension (10) :: classes
end type Student
```

The `Student` type here contains exactly one component of type `Personnel`, as well as two additional members needed to describe student records, an integer `nclasses` which contains the number of classes a student is enrolled in, and an array of 10 fixed length characters called `classes` for the names of those classes. In C++, the component corresponding to type `Personnel` is implicit and would not be declared.

To initialize the `Student` type, one can call the constructor for `Personnel` to initialize the `Personnel` component of `Student` and initialize the other components by direct assignment, as follows:

```
type (Student) :: studentA
call new(studentA%personnel,0,'PAT','SMITH')
studentA%nclasses = 0
```

These operations can be incorporated into a `Student` constructor procedure:

```
subroutine init_Student(this,s,fn,ln)
! Student class constructor
type (Student), intent (out) :: this
integer, intent (in) :: s
character*(*), intent (in) :: fn, ln
call new(this%personnel,s,fn,ln)
this%nclasses = 0
end subroutine init_Student
```

which emulates the initialization list which occurs in C++. In a similar fashion, the `Personnel` component of `Student` can be deleted by applying the `Personnel` destructor:

```
call delete(this%personnel)
```

A Student destructor can be written to execute this operation. In C++, a destructor which deletes only the inherited data member of a derived class does not have to be explicitly created.

Thus a primitive Student class which builds upon the Personnel class can be constructed as follows:

```

    module Student_class
! bring Personnel_class into scope
    use Personnel_class
    private :: init_Student, term_Student
! define Student type
    type Student
    private
        type (Personnel) :: personnel
        integer :: nclasses
        character*12, dimension (10) :: classes
    end type Student
    interface new
        module procedure init_Student
    end interface
    interface delete
        module procedure term_Student
    end interface
    contains
        subroutine init_Student(this,s,fn,ln)
! Student class constructor
        type (Student), intent (out) :: this
        integer, intent (in) :: s
        character*(*), intent (in) :: fn, ln
        call new(this%personnel,s,fn,ln)
        this%nclasses = 0
        end subroutine init_Student
!
        subroutine term_Student(this)
! Student class destructor
        type (Student), intent (inout) :: this
        call delete(this%personnel)
        end subroutine term_Student
    end module Student_class

```

Here the USE PERSONNEL\_CLASS statement plays the role of the class derivation list in C++. We have also overloaded the names new and delete so that they execute the Student constructor and destructor if the argument is of type Student. With this incomplete class one can create and destroy a Student record as follows:

```

program student_test
use Student_class
type (Student) :: studentA
call new(studentA,0, 'PAT', 'SMITH')
call delete(studentA)
end program student_test

```

Inheritance of methods is constructed by having the derived class procedure delegate to the base class. This is a common approach in C++ when methods have to be modified, but in Fortran90 it is required even when methods are not modified. For example, to print a Student record, one would delegate to the Personnel class the responsibility for printing out the Personnel component of Student, as follows:

```

call print(studentA%personnel)

```

In C++, one would have to use the scope operator:

```

Personnel::studentA.print();

```

If the print procedure for a Student is modified to also print out the enrollment record, this delegation is incorporated into the modified procedure, just as in C++. The following is an example of such a modified procedure:

```

subroutine print_Student(this,printssn)
type (Student), intent (in) :: this
logical, optional, intent (in) :: printssn
integer :: i, j
! delegate printing of personnel component
call print(this%personnel,printssn)
! print enrollment record
if (this%nclasses==0) then
  Print *, '-- Not Enrolled'
else
  Print *, '-- Enrolled'
  do i = 1, this%nclasses
    do j = 1, size(this%classes(i)%stringptr)
      write (*, '(a)', advance='no')
&      this%classes(i)%stringptr(j)
    enddo
  enddo
  Print *
endif
end subroutine print_Student

```

Finally, if we overload the name print with the INTERFACE statement to include the print\_Student procedure, the name print will execute the correct procedure for Student objects.

Since the print function in the base class was modified for the derived class, the process of creating the modified version is similar to what one might do in C++. However, for functions not modified in the derived class, nothing needs to be done in C++, whereas in Fortran90 one needs to write a procedure which delegates to the base class the

responsibility for carrying out the procedure on behalf of the derived class. For example, the procedure `getssn_Student` needs to be created in Fortran90 which would not have to be created in C++:

```
integer function getssn_Student(this)
type (Student), intent (in) :: this
getssn_Student = getssn_Personnel(this%personnel)
end function getssn_Student
```

If the components of `Personnel` had been `PUBLIC`, one could have accessed the identifier directly instead of using the `getssn_Personnel` procedure, as follows:

```
getssn_Student = this%personnel%ssn
```

In order for the `Student` class to be useful, we create a procedure to add a class to the student's file:

```
subroutine addclass(this,c)
! Add a class to a student file
type (Student), intent (inout) :: this
character(*), intent (in) :: c
this%nclasses = this%nclasses + 1
this%classes(this%nclasses) = c
end subroutine addclass
```

In this procedure, the class data member `nclasses` is incremented, and the name of the class (contained in the argument `c`) is added to the next element of the array `classes`. The following main program tests this class:

```
program student_test
use Student_class
! create a record
type (Student) :: studentA
call new(studentA,0,'PAT','SMITH')
call addclass(studentA,'MATH')
! print a record
call print(studentA,.true.)
end program student_test
```

and produces the following output:

```
0 : PAT SMITH
-- Enrolled
MATH
```

What we have constructed here is an inheritance hierarchy which does not contain virtual functions. We have written a new class which contains the data of the base class and all the procedures of the base class have been extended to work with the new derived class. `INTERFACE` statements have to be used to give the procedure uniform names. Note that the derived class did not need to know how the base class was implemented. What is missing here is dynamic dispatching (or run-time polymorphism), which will be discussed in

the next section.

As before, it is simpler to make the `Student` data member `classes` an array of character variables of fixed size. It is possible, however, to make this an allocatable array of pointers to C-style character arrays as Henderson and Zorn do, but one must do so indirectly. In Fortran90, a pointer is actually an attribute and not a data type, so it is impossible to create an array of pointers directly. Instead, one creates a derived type which contains a pointer and then creates an array of that derived type. Thus we can create a type called `String` as follows:

```
type, private :: String
  character*1, dimension(:), pointer :: stringptr
end type String
```

And then redefine the `Student` type to contain an array of `Strings`:

```
type Student
  type (Personnel) :: personnel
  integer :: nclasses
  type (String), dimension (10) :: classes
end type Student
```

If `studentA`'s first class is math, one can allocate memory and assign the `classes` data member, as follows:

```
allocate(studentA%classes(1)%stringptr(len('MATH')))
call strcpy(studentA%classes(1)%stringptr,'MATH')
```

To use such an array of pointers, a similar allocation and assignment must be added to the `addclass` procedure. Similarly, a `do` loop must be added to the destructor for the class to allow deallocation of memory:

```
do i = 1, studentA%nclasses
  deallocate(studentA%classes(i)%stringptr)
enddo
```

As in C++, it is possible to hide some of the grungy details of C-style string manipulation by creating a special string class in Fortran90. The final version of the `Student` class which uses an array of pointers is listed in Appendix B.

Objects of the `Personnel` class were not intended to be created. The usual way to enforce this in C++ is to make `Personnel` an abstract base class. Henderson and Zorn did not do so because they wanted to implement methods common to the hierarchy in this class. An alternative way to enforce this in C++ is to make the `Personnel` constructor protected. The protected keyword is not available in Fortran90, but its effect can be partially emulated by declaring `PUBLIC` the name `Personnel` (rather than the constructor) in the `Personnel` class, but then declaring it `PRIVATE` in a derived class such as `Student`. Then any program which "uses" the `Student` class will not have access to the `Personnel` type. However, the emulation is incomplete, since a program unit which "uses" the `Personnel` class directly will have access to the `Personnel` type.

In a similar manner to the `Student` class, one can derive another class from `Personnel`, called `Teacher`. The `Teacher` type will contain a new member called `salary`:

```

type Teacher
  private
  type (Personnel) :: personnel
  integer :: salary
end type Teacher

```

We will add to this class a new procedure called `updatesalary` to update the salary data member. The `print` procedure for `Teacher` is also modified to print the salary. The final version of the `Teacher` class is listed in Appendix C. A following program tests `Student` and `Teacher` objects:

```

program records_test
  use Student_class
  use Teacher_class
! create records
  type (Student) :: studentA
  type (Teacher) :: teacherA
  call new(studentA,0,'PAT','SMITH')
  call new(teacherA,2,'JOHN','WHITE',1000)
  call addclass(studentA,'MATH')
  call updatesalary(teacherA,2000)
! print records
  call print(studentA,.true.)
  call print(teacherA,.false.)
  Print *, 'NUM_FILES=',get_num_files()
! delete records
  call delete(teacherA)
  call delete(studentA)
end program records_test

```

and produces the following result:

```

0 : PAT SMITH
-- Enrolled
MATH
JOHN WHITE
-- Salary: 2000
NUM_FILES= 2

```

## IV. Dynamic Dispatching

A third important concept in C++ which must be modeled is the idea of dynamic dispatching or run-time polymorphism. In the previous section on inheritance, we showed how a single method name could respond differently to different objects in an inheritance hierarchy. Dynamic dispatching allows a single object name to refer to any member of an inheritance hierarchy and permits a procedure to resolve at run-time which actual object is being referred to. This ability is useful because it allows one to write a generic program for a whole class of related objects, yet have the program behave differently depending on the object being used.

To implement dynamic dispatching in Fortran90, two features must be constructed: first, a pointer object which can point to any member in an inheritance hierarchy, and second, a dispatch mechanism (or method lookup) which can select the appropriate procedure (method) to execute based on the actual class referenced in the pointer object. In C++ these features are present automatically through the use of virtual functions. In Fortran90 they will be constructed by implementing a polymorphic class. Although the details of dynamic dispatching are exposed to the writer of this class, they can be hidden from the procedures which make use of this class, as we will show in the next section.

A pointer object can be created for our `Personnel` class by defining a `poly_Personnel` type, as follows:

```
type poly_Personnel
  type (Student), pointer :: ps
  type (Teacher), pointer :: pt
end type poly_Personnel
```

This type definition contains pointers to all the possible types in the inheritance hierarchy. We have omitted the `Personnel` type from this list because it was intended to be an abstract type without concrete objects. At any given time, we will associate one of the pointers in this list with an actual object, and the other pointers will be set to null objects. To illustrate how this works, the following program fragment will create an object called `person` of type `poly_Personnel`, and then assign a `Student` object to `person` as follows:

```
type (Student), target :: studentA
type (poly_Personnel) :: person
call new(studentA,0,'PAT','SMITH')
! assign student object to polymorphic object
person%ps => studentA
! nullify other possibilities
nullify(person%pt)
```

Fortran90 uses the `'=>'` operator to assign pointers to objects, and objects being pointed at must have the `TARGET` attribute. Since the internal state of a pointer in Fortran90 is private, the `NULLIFY` intrinsic is needed to set it to a null object. This assignment operation can be encapsulated into a procedure as follows:

```
function assign_student(ps) result(pps)
type (poly_Personnel) :: pps
type (Student), target, intent(in) :: ps
pps%ps => ps
nullify(pps%pt)
```

```
end function assign_student
```

Thus, one can create a Student object and assign it to person as follows:

```
call new(studentA,0,'PAT','SMITH')
person = assign_student(studentA)
```

In a similar fashion one can create an assignment procedure for a Teacher.

The second feature that we must construct is a dispatch mechanism to select the appropriate procedure to execute. This is done by checking which of the possible pointers actually points to an object and then passing the associated pointer to the appropriate procedure. In Fortran90, the ASSOCIATED intrinsic is used for this purpose as follows:

```
if (associated(person%ps)) Print *,'We have a Student!'
```

Thus one can write a print procedure for objects of type poly\_Personnel which checks which type has been associated and executes the appropriate procedure, as follows:

```
subroutine poly_print(this)
  type (poly_Personnel), intent (in) :: this
! check if pointer is associated with student type
  if (associated(this%ps)) then
    call print(this%ps)
! check if pointer is associated with teacher type
  elseif (associated(this%pt)) then
    call print(this%pt)
  endif
end subroutine poly_print
```

Finally, we can overload the name poly to refer to the assign\_student and assign\_teacher procedures, and the name print to refer to poly\_print. All these features can be combined into a simple poly\_Personnel class, as follows:

```

    module poly_Personnel_class
! bring Student_class into scope
    use Student_class
! bring Teacher_class into scope
    use Teacher_class
    private :: assign_student, assign_teacher, poly_print
! define poly_Personnel_type
    type poly_Personnel
        private
            type (Student), pointer :: ps
            type (Teacher), pointer :: pt
        end type poly_Personnel
    interface poly
        module procedure assign_student, assign_teacher
    end interface
    interface print
        module procedure poly_print
    end interface
    contains
        function assign_student(ps) result(pps)
! assign Student to poly_Personnel
            type (poly_Personnel) :: pps
            type (Student), target, intent(in) :: ps
            pps%ps => ps
            nullify(pps%pt)
        end function assign_student
!
        function assign_teacher(pt) result(pps)
! assign Teacher to poly_Personnel
            type (poly_Personnel) :: pps
            type (Teacher), target, intent(in) :: pt
            nullify(pps%ps)
            pps%pt => pt
        end function assign_teacher
!
        subroutine poly_print(this,printssn)
! Print poly_Personnel
            type (poly_Personnel), intent (in) :: this
            logical, optional, intent (in) :: printssn
            if (associated(this%ps)) then
                call print(this%ps,printssn)
            elseif (associated(this%pt)) then
                call print(this%pt,printssn)
            endif
        end subroutine poly_print
    end module poly_Personnel_class

```

In the following sample program, the object person functions as a pointer to base class objects which can be assigned either to a Student or a Teacher object and be passed to the appropriate print procedure :



```

    program poly_test
! bring in poly_Personnel_class into scope
    use poly_Personnel_class
    type (Student), target :: studentA
    type (Teacher), target :: teacherA
    type (poly_Personnel) :: person
! initialize student and teacher
    call new(studentA,0,'PAT','SMITH')
    call new(teacherA,2,'JOHN','WHITE',1000)
! assign a student to person and print record
    person = poly(studentA)
    call print(person,.true.)
! assign a teacher to person and print record
    person = poly(teacherA)
    call print(person,.false.)
    end program poly_test

```

This program produces the following output:

```

0 : PAT SMITH
-- Not Enrolled
JOHN WHITE
-- Salary: 1000

```

A more complete version of this polymorphic class is listed in Appendix D, where a constructor as well as dynamically dispatched versions of the remaining procedures in the Personnel hierarchy have been implemented (`getssn`, `addclass`, and `updatesalary`). The polymorphic class emulates the virtual function mechanism in C++. Note that the polymorphic class knows only about the types and interfaces in the hierarchy and nothing about their implementation. This makes writing a polymorphic class rather mechanical and it could in principle be done by a software tool.

## V. Database Application with Dynamic Dispatching

Henderson and Zorn make use of the `Personnel` class hierarchy to write a `Database` class which manages a linked list of `Personnel` objects. The class data members for this class contain a pointer to the base class object and a pointer to the `Database` object. This is expressed by the following Fortran90 type:

```
type Database
  type (poly_Personnel) :: file
  type (Database), pointer :: next
end type Database
```

where the `poly_Personnel` component is used instead of the pointer to `Personnel`. The `Database` class contains methods to add, remove, locate and print records in the database. These methods are written much the same as one would write them in C++, except for the use of `poly_Personnel`. For example, a method to add a file to the database looks like:

```
subroutine add(this,f)
  type (Database), target, intent (inout) :: this
  type (poly_Personnel) :: f
  type (Database), pointer :: tmp
  tmp => this
! traverse database
  do while (associated(tmp%next))
    tmp => tmp%next
  enddo
! store record in current location
  tmp%file = f
! allocate next location
  allocate(tmp%next)
  call new(tmp%next)
end subroutine add
```

This procedure traverses the `Database` pointers until a null `next` pointer is found, and then it stores the file record `f` in the current location and allocates the `next` location. In Fortran90 the `%` syntax is used both for pointer as well as object components, whereas in C++ the `->` syntax is used for pointers and the `.` for objects. Note that the `Database` argument `'this'` requires a `TARGET` attribute in Fortran90 to allow it to be pointed at. To print a database, one traverses it in a similar manner, printing each valid record. To remove or locate a record from the database, one searches the database for a particular identifier, then deletes or returns it. For example, the procedure for returning a record from the database looks like:

```

        type (poly_Personnel) function locate(this,s)
        type (Database), target, intent (in) :: this
        integer, intent (in) :: s
        type (Database), pointer :: tmp
        tmp => this
! traverse database
        do while (associated(tmp%next))
! check identifier
            if (getssn(tmp%file)==s) then
! return record
                locate = tmp%file
                return
            endif
            tmp => tmp%next
        enddo
    end function locate

```

The remaining procedures in the Database class are listed in Appendix E. The following test program will first create a database and add a student and teacher record to it. Then it will retrieve a student file from the database and add a physics class, and retrieve a teacher file and update the salary. Finally, it will print the entire database (without identifiers) and purge it. The C++ version of the test program is listed in Appendix F.

```

    program database_test
! bring Database_class into scope
    use Database_class
    implicit none
    integer :: i
    type (Database), target :: cs
    type (poly_Personnel) :: person
    type (Student), pointer :: pstudent
    type (Teacher), pointer :: pteacher
! Initialize database
    call new(cs)
! Add a student file
    allocate(pstudent)
    call new(pstudent,1,'PAUL','JONES')
    person = poly(pstudent)
    call add(cs,person)
! Add a teacher file
    allocate(pteacher)
    call new(pteacher,2,'JOHN','WHITE',1000)
    person = poly(pteacher)
    call add(cs,person)
! Locate item in the database with ssn = 1
    person = locate(cs,1)
! Add a physics class
    call addclass(person,'PHYSICS')
! Locate item in the database with ssn = 2
    person = locate(cs,2)
! Update the salary
    call updatesalary(person,2000)
! Print the database
    call print(cs)
! Delete each data file from database
    do i = 1, get_num_files()
        call remove(cs,i)
    enddo
    end program database_test

```

The output of this program is:

```

PAUL JONES
-- Enrolled
PHYSICS
JOHN WHITE
-- Salary: 2000

```

## VI. Multiple Inheritance, Templates, and Operators

Multiple inheritance in C++ allows one to create composite classes that have the properties of its base classes. An example of such a class might be a `StudentTeacher`, which inherits from both `Student` and `Teacher`. In Fortran90, one might implement such a composite class with the following composite type:

```
type StudentTeacher
  type (Student) :: student
  type (Teacher) :: teacher
end type StudentTeacher
```

which includes a component of `Student` and a component of `Teacher`. This type is included in a module and the `USE` operator is used to bring the base classes into scope.

```
module StudentTeacher_class
! bring Student_class and Teacher_class into scope
  use Student_class
  use Teacher_class
! define StudentTeacher type
  type StudentTeacher
    type (Student) :: student
    type (Teacher) :: teacher
  end type StudentTeacher
end module StudentTeacher_class
```

One also implements the class member functions in the usual way, by delegating the operation on the `Student` component of `StudentTeacher` to the `Student` member function and similarly for the `Teacher` component. Notice that both `Student` and `Teacher` each contain a component of `Personnel`, which is now multiply defined. Thus when the `print` function is invoked, the name of the `StudentTeacher` object will be printed twice. If the `StudentTeacher` is really only one person (for example, a Teaching Assistant at a university), this is not the desired behavior. C++ has the mechanism of a virtual base class to eliminate duplication of inherited class data members. In Fortran90, since the base class data members are declared explicitly, one can create a `StudentTeacher` consisting of a single `Personnel` component with all the additional components which belong to a `Student` and a `Teacher`, as follows:

```
type StudentTeacher
  type (Personnel) :: personnel
  integer :: nclasses
  type (String), dimension (10) :: classes
  integer :: salary
end type StudentTeacher
```

Alternatively, a `StudentTeacher` could consist of a single `Student` component with the additional component which belongs to a `Teacher`, as follows:

```
type StudentTeacher
  type (Student) :: student
  integer :: salary
```

```
end type StudentTeacher
```

Thus emulating multiple inheritance in Fortran90 poses no more difficulty than implementing single inheritance.

Templates are an important new feature in C++ which allows one to write procedures in terms of a parametrized type which can be instantiated with multiple actual types. This allows one to write generic functions that are independent of type. In contrast to inheritance, templates avoid replication of source code, but not of executable code.

Fortran90 has no mechanism for templates or parametrized types, and we know of no effective way to emulate them directly. However, some of the functionality of templates can be achieved by use of the polymorphic class described in section IV. Since the polymorphic type we constructed can consist of any types, not just those related by inheritance, it is possible to write one function which can be used with different actual types.

Fortran90 allows only a limited number of operators to be overloaded. Notably missing are subscript [ ] and call operators ( ). Fortran90 does allow one, however, to create a generic user-defined operator of the form .USER\_OP\_NAME. which can be used as either a binary or unary operator. For example, the increment operator

```
a++
```

can be implemented as:

```
.increment.a
```

In Fortran90, none of these operators can appear on the left hand side of an assignment.

## VII. Conclusions

Fortran90 is able to express many of the important concepts of C++, such as abstract data types, encapsulation, function overloading, and classes directly. Concepts such as inheritance are not directly supported, but can be emulated. For functions in a derived class which are modified, the procedure is similar to what one would do in C++, except that INTERFACE statements are needed in Fortran90 to allow procedures in different classes to have the same names. In contrast to C++, procedures which are not modified must also be created in Fortran90 to delegate the method to the base class. For classes without virtual functions, this emulation is quite straightforward and is tedious only if the inheritance hierarchy contains many unmodified functions. The emulation of dynamic dispatching is more involved and requires the creation of a polymorphic class. This class contains an object which can point to any member of the inheritance hierarchy and a generic method for each class member function which can dynamically determine which actual function to execute. Writing the polymorphic class is straightforward, but can be tedious, especially if the inheritance hierarchy is deep. The details of this class can be encapsulated, however, so that it can be used by other classes without concern for how dynamic dispatching is implemented, just as in C++. Implementing multiple inheritance introduces no new concepts. Templates or parametrized types are not supported in Fortran90, and no effective way has yet been found to emulate them. The Fortran90 programmer must provide explicitly many features which are automatically available in C++. This can be enlightening for a beginner in OOP, but can be tedious for the advanced practitioner.

The C++ language is very powerful, flexible and complex. It is a language which is constantly evolving with new ideas. It is relatively poor in standard libraries and intrinsics, although that may improve with the adoption of the Standard Template Library. Fortran90 is a more conservative, stable language, rich with many intrinsics useful for scientific programming. Scientific programmers are caught in a bind. They often do not want to be on the “bleeding edge” of programming languages. Yet they want to adopt useful, proven programming methods. Does Fortran90 go far enough in introducing new methodology to Fortran? The answer is a subject of debate and is dependent on the problem being modeled.

In our own experience in implementing object-oriented plasma simulation codes in both C++ and Fortran90 [6], we found benefits and drawbacks in each language. In the C++ version of the codes, we had to create special classes to obtain the use of self-describing multi-dimensional arrays, which were automatically available in Fortran90. On the other hand, special polymorphic classes had to be created in Fortran90 to emulate dynamic dispatching, which was automatically available in C++. Because type checking in Fortran90 was more strict, more errors were caught by the compiler than in C++ and debugging went more quickly. However, one had to write more code. As expected for a mature language, the Fortran90 environment was very stable and uniform across platforms. And not surprising for an evolving language, the environment of C++ varied across vendors and platforms and some new features (such as templates) were sometimes poorly implemented. The Fortran90 version of the code executed about twice as fast as the C++ version.

There are many kinds of relationships between classes which can occur in object-oriented programming [7]. In addition to inheritance (is-a), there are aggregations (is-part-of, or has-a) and links (is-connected-to, or serves). The relative importance of these various relationships depends on the problem domain. For problems with many objects which are almost identical, such as modeling power supplies [8], the resulting deep inheritance hierarchy would be very tedious to model in Fortran90, although possible. For problems which are dominated by aggregations and links, such as plasma simulation, Fortran90 is as

expressive as C++.

#### Acknowledgments:

The research of Viktor K. Decyk was carried out in part at UCLA and was sponsored by USDOE and NSF. It was also carried out in part at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The research of Charles D. Norton was supported by a National Research Council Associateship, and that of Boleslaw K. Szymanski was sponsored under grants CCR-9216053 and CCR-9527151. We would like to thank R. Henderson and B. Zorn for making their source code available and Chris Myers for helpful discussions and suggestions about this manuscript.

## References:

- [1] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen, Object-Oriented Modeling and Design [Prentice-Hall, Englewood Cliffs, NJ, 1991], chapter 16.
- [2] Stanley B. Lippman, C++ Primer, [Addison-Wesley, Reading, Massachusetts, 1991].
- [3] T. M. R. Ellis, Ivor R. Philips, and Thomas M. Lahey, Fortran 90 Programming, [Addison-Wesley, Reading, Massachusetts, 1994].
- [4] V. K. Decyk, C. D. Norton, and B. K. Szymanski, "Introduction to Object-Oriented Concepts using Fortran90," UCLA IPFR Report PPG-1560, July, 1996. See also the web site: <http://www.cs.rpi.edu/~szymansk/oof90.html>.
- [5] R. Henderson and B. Zorn, "A Comparison of Object-Oriented Programming in Four Modern Languages," Software-Practice and Experience, Vol. 24, Num. 11, pp. 1077-1095, Nov. 1994.
- [6] C. D. Norton, V. K. Decyk, and B. K. Szymanski, "High Performance Object Oriented Scientific Programming in Fortran 90," Proc. Eighth SIAM Conf. on Parallel Processing for Scientific Computing, Minneapolis, MN, 1997.
- [7] Grady Booch, Object-Oriented Analysis and Design [Benjamin/Cummings, Redwood City, CA, 1994], chapter 3.
- [8] John J. Barton and Lee R. Nackman, Scientific and Engineering C++ [Addison Wesley, Reading, Massachusetts, 1994], chapter 9.

## Appendix A: Final version of Personnel class

### Fortran90 version

```
module Personnel_class
  implicit none
  private :: init_Personnel, term_Personnel, print_Personnel
! Define Personnel type
  type Personnel
    private
    integer :: ssn
    character, dimension(:), pointer :: firstname, lastname
  end type Personnel
! Number of database records
  integer, save, private :: NUM_FILES = 0
  interface new
    module procedure init_Personnel
  end interface
  interface delete
    module procedure term_Personnel
  end interface
  interface print
    module procedure print_Personnel
  end interface
  contains
    subroutine init_Personnel(this,s,fn,ln)
! Constructor
      type (Personnel), intent (out) :: this
      integer, intent (in) :: s
      character*(*), intent (in) :: fn, ln
      this%ssn = s
      allocate(this%firstname(len(fn)),this%lastname(len(ln)))
      call strcpy(this%firstname,fn)
      call strcpy(this%lastname,ln)
      NUM_FILES = NUM_FILES + 1
    end subroutine init_Personnel
!
      subroutine term_Personnel(this)
! Destructor
      type (Personnel), intent (inout) :: this
      deallocate(this%firstname,this%lastname)
      NUM_FILES = NUM_FILES - 1
    end subroutine term_Personnel
```

```

subroutine print_Personnel(this,printssn)
type (Personnel), intent (in) :: this
logical, optional, intent (in) :: printssn
if (present(printssn)) then
    if (printssn) write (*,'(i2,a2)',advance='no')
&    this%ssn,': '
endif
Print *, this%firstname, ' ', this%lastname
end subroutine print_Personnel
!

function getssn_Personnel(this) result(ssn)
type (Personnel), intent (in) :: this
integer :: ssn
ssn = this%ssn
end function getssn_Personnel
!

integer function get_num_files()
get_num_files = NUM_FILES
end function get_num_files
!

subroutine strcpy(s,c)
character, dimension (:), intent (out) :: s
character*(*), intent (in) :: c
integer :: i
do i = 1, max(size(s),len(c))
    s(i) = c(i:i)
enddo
end subroutine strcpy
end module Personnel_class

```

## C++ version

```
// *****  
// **** personnel.h ****  
// *****  
  
#include <stream.h>  
  
class Personnel {  
    static int NUM_FILES;  
    char *firstname, *lastname;  
protected:  
    int ssn;  
public:  
    Personnel(const int s, const char *fn, const char *ln);  
    ~Personnel();  
    virtual void print(const int printssn = 0);  
    virtual int getssn();  
    static int get_num_files();  
};
```

```

//*****
//****  personnel.cc  ****
//*****

#include <stream.h>
#include <string.h>
#include "personnel.h"

// Initialize static class member
// Number of database records
int Personnel::NUM_FILES = 0;

// Constructor
Personnel::Personnel(const int s, const char *fn, const char *ln)
{
    ssn = s;
    firstname = new char[strlen(fn)+1];
    lastname = new char[strlen(ln)+1];
    strcpy(firstname, fn);
    strcpy(lastname, ln);
    NUM_FILES++;
}

// Destructor
Personnel::~Personnel()
{
    delete firstname;
    delete lastname;
    NUM_FILES--;
}

void Personnel::print(const int printssn)
{
    if (printssn)
        cout << ssn << ": " << firstname << ' ' << lastname << endl;
    else
        cout << firstname << ' ' << lastname << endl;
}

int Personnel::getssn() { return ssn; }

int Personnel::get_num_files() { return NUM_FILES; }

```

## Appendix B: Final version of Student class

### Fortran90 version

```
module Student_class
! bring Personnel_class into scope
  use Personnel_class
  implicit none
  private :: Personnel,init_Student,term_Student,print_Student
  private :: getssn_Personnel, getssn_Student
! define String type
  type, private :: String
    character*1, dimension(:), pointer :: stringptr
  end type String
! define Student type
  type Student
  private
    type (Personnel) :: personnel
    integer :: nclasses
    type (String), dimension (10) :: classes
  end type Student
  interface new
    module procedure init_Student
  end interface
  interface delete
    module procedure term_Student
  end interface
  interface print
    module procedure print_Student
  end interface
  interface getssn
    module procedure getssn_Student
  end interface
  contains
    subroutine init_Student(this,s,fn,ln)
! Student class constructor
    type (Student), intent (out) :: this
    integer, intent (in) :: s
    character*(*), intent (in) :: fn, ln
    call new(this%personnel,s,fn,ln)
    this%nclasses = 0
  end subroutine init_Student
!
    subroutine term_Student(this)
! Student class destructor
    type (Student), intent (inout) :: this
    integer :: i
    call delete(this%personnel)
    do i = 1, this%nclasses
      deallocate(this%classes(i)%stringptr)
    enddo
  end subroutine term_Student
end module Student_class
```

```
end subroutine term_Student
```

```

        subroutine print_Student(this,printssn)
! Print a student file
        type (Student), intent (in) :: this
        logical, optional, intent (in) :: printssn
        integer :: i, j
        call print(this%personnel,printssn)
        if (this%nclasses==0) then
            Print *, '-- Not Enrolled'
        else
            Print *, '-- Enrolled'
            do i = 1, this%nclasses
                do j = 1, size(this%classes(i)%stringptr)
                    write (*, '(a)', advance='no')
&                 this%classes(i)%stringptr(j)
                enddo
            enddo
            Print *
        endif
    end subroutine print_Student

!

        integer function getssn_Student(this)
        type (Student), intent (in) :: this
        getssn_Student = getssn_Personnel(this%personnel)
    end function getssn_Student

!

        subroutine addclass(this,c)
! Add a class to a student file
        type (Student), intent (inout) :: this
        character*(*), intent (in) :: c
        this%nclasses = this%nclasses + 1
        allocate(this%classes(this%nclasses)%stringptr(len(c)))
        call strcpy(this%classes(this%nclasses)%stringptr,c)
    end subroutine addclass
end module Student_class

```

C++ version

```
// *****  
// **** student.h ****  
// *****  
  
class Student : public Personnel {  
    int nclasses;  
    char *classes[10];  
public:  
    Student(const int ssn, const char *firstname, const char  
*lastname);  
    ~Student();  
    void print(const int printssn = 0);  
    void addclass(const char *c);  
};
```

```

//*****
//**** student.cc ****
//*****

#include <stream.h>
#include <string.h>

#include "personnel.h"
#include "student.h"

// Student class constructor
Student::Student(const int s, const char *fn, const char *ln) :
Personnel(s,fn,ln)
{
    nclasses=0;
}

// Student class destructor
Student::~~Student()
{
    for (int i=0; i<nclasses; ++i) delete classes[i];
}

// Add a class to a student file
void Student::addclass(const char *c)
{
    classes[nclasses] = new char[strlen(c)+1];
    strcpy(classes[nclasses], c);
    nclasses += 1;
}

// Print a student file
void Student::print(const int printssn)
{
    Personnel::print(printssn);
    if (nclasses == 0)
        cout << "-- Not Enrolled" << endl;
    else {
        cout << "-- Enrolled:" << endl;
        for (int i=0; i < nclasses; ++i) cout << classes[i];
        cout << endl;
    }
}
}

```

## Appendix C: Final version of Teacher class

### Fortran90 version

```
module Teacher_class
! bring Personnel_class into scope
  use Personnel_class
  implicit none
  private :: Personnel,init_Teacher,term_Teacher,print_Teacher
  private :: getssn_Personnel, getssn_Teacher
! define Teacher type
  type Teacher
  private
    type (Personnel) :: personnel
    integer :: salary
  end type Teacher
  interface new
    module procedure init_Teacher
  end interface
  interface delete
    module procedure term_Teacher
  end interface
  interface print
    module procedure print_Teacher
  end interface
  interface getssn
    module procedure getssn_Teacher
  end interface
  contains
    subroutine init_Teacher(this,s,fn,ln,sal)
! Teacher constructor
      type (Teacher), intent (out) :: this
      integer, intent (in) :: s, sal
      character*(*), intent (in) :: fn, ln
      call new(this%personnel,s,fn,ln)
      this%salary = sal
    end subroutine init_Teacher
!
    subroutine term_Teacher(this)
! Teacher class destructor
      type (Teacher), intent (inout) :: this
      call delete(this%personnel)
    end subroutine term_Teacher
!
    subroutine print_Teacher(this,printssn)
! Print a teacher file
      type (Teacher), intent (in) :: this
      logical, optional, intent (in) :: printssn
      call print(this%personnel,printssn)
      Print *, '-- Salary: ', this%salary
    end subroutine print_Teacher
```



```
integer function getssn_Teacher(this)
type (Teacher), intent (in) :: this
getssn_Teacher = getssn_Personnel(this%personnel)
end function getssn_Teacher
!
subroutine updatesalary(this,sal)
type (Teacher), intent (inout) :: this
integer, intent (in) :: sal
this%salary = sal
end subroutine updatesalary
end module Teacher_class
```

## C++ version

```
// *****
// ****  teacher.h  ****
// *****

class Teacher : public Personnel {
    int salary;
public:
    Teacher(const int ssn, const char *firstname,
            const char *lastname, const int salary);
    void print(const int printssn = 0);
    void updatesalary(const int sal);
};

//*****
//****  teacher.cc  ****
//*****

#include <stream.h>

#include "personnel.h"
#include "teacher.h"

// Teacher constructor
Teacher::Teacher(const int s, const char *fn, const char* ln,
                const int sal) : Personnel(s,fn,ln)
{
    salary = sal;
}

// Print a teacher file
void Teacher::print(const int printssn)
{
    Personnel::print(printssn);
    cout << "-- Salary: " << salary << endl;
}

void Teacher::updatesalary(const int sal)
{
    salary = sal;
}
```

## Appendix D: Final version of poly\_Personnel class

### Fortran90 version

```
module poly_Personnel_class
! bring Student_class into scope
  use Student_class
! bring Teacher_class into scope
  use Teacher_class
  private :: poly_init, assign_student, assign_teacher
  private :: poly_print, poly_getssn, poly_addclass
  private :: poly_updatesalary
! define poly_Personnel type
  type poly_Personnel
    private
      type (Student), pointer :: ps
      type (Teacher), pointer :: pt
  end type poly_Personnel
  interface new
    module procedure poly_init
  end interface
  interface poly
    module procedure assign_student, assign_teacher
  end interface
  interface print
    module procedure poly_print
  end interface
  interface getssn
    module procedure poly_getssn
  end interface
  interface addclass
    module procedure addclass, poly_addclass
  end interface
  interface updatesalary
    module procedure updatesalary, poly_updatesalary
  end interface
  contains
    subroutine poly_init(this)
! Initialize poly_Personnel with null pointers
      type (poly_Personnel), intent (out) :: this
      nullify(this%ps)
      nullify(this%pt)
    end subroutine poly_init
!
    function assign_student(ps) result(pps)
! assign Student to poly_Personnel
      type (poly_Personnel) :: pps
      type (Student), target, intent(in) :: ps
      pps%ps => ps
      nullify(pps%pt)
    end function assign_student
```

```

        function assign_teacher(pt) result(pps)
! assign Teacher to poly_Personnel
        type (poly_Personnel) :: pps
        type (Teacher), target, intent(in) :: pt
        nullify(pps%ps)
        pps%pt => pt
        end function assign_teacher
!
        subroutine poly_print(this,printssn)
! Print poly_Personnel
        type (poly_Personnel), intent (in) :: this
        logical, optional, intent (in) :: printssn
        if (associated(this%ps)) then
            call print(this%ps,printssn)
        elseif (associated(this%pt)) then
            call print(this%pt,printssn)
        endif
        end subroutine poly_print
!
        integer function poly_getssn(this)
        type (poly_Personnel), intent (in) :: this
        if (associated(this%ps)) then
            poly_getssn = getssn(this%ps)
        elseif (associated(this%pt)) then
            poly_getssn = getssn(this%pt)
        endif
        end function poly_getssn
!
        subroutine poly_addclass(this,c)
! Add a class to a student poly_Personnel file
        type (poly_Personnel), intent (inout) :: this
        character*(*), intent (in) :: c
        if (associated(this%ps)) call addclass(this%ps,c)
        end subroutine poly_addclass
!
        subroutine poly_updatesalary(this,sal)
        type (poly_Personnel), intent (inout) :: this
        integer, intent (in) :: sal
        if (associated(this%pt)) call updatesalary(this%pt,sal)
        end subroutine poly_updatesalary
end module poly_Personnel_class

```

## Appendix E: Final version of Database class

### Fortran90 version

```
module Database_class
! bring poly_Personnel_class into scope
  use poly_Personnel_class
  private :: init_Database, print_Database
! define Database type
  type Database
  private
    type (poly_Personnel) :: file
    type (Database), pointer :: next
  end type Database
  interface new
    module procedure init_Database
  end interface
  interface print
    module procedure print_Database
  end interface
  contains
    subroutine init_Database(this)
! Constructor
      type (Database), intent (out) :: this
! nullify pointers
      call new(this%file)
      nullify(this%next)
    end subroutine init_Database
!
    subroutine print_Database(this)
! Print the database
      type (Database), target, intent (inout) :: this
      type (Database), pointer :: tmp
      tmp => this
      do while (associated(tmp%next))
        call print(tmp%file)
        tmp => tmp%next
      enddo
    end subroutine print_Database
end module Database_class
```

```

        subroutine add(this,f)
! Add a file to the database
        type (Database), target, intent (inout) :: this
        type (poly_Personnel) :: f
        type (Database), pointer :: tmp
        tmp => this
        do while (associated(tmp%next))
            tmp => tmp%next
        enddo
        tmp%file = f
        allocate(tmp%next)
        call new(tmp%next)
        end subroutine add

!

        subroutine remove(this,s)
! Remove a file from the database
        type (Database), target, intent (inout) :: this
        integer, intent (in) :: s
        type (Database), pointer :: tmp
        tmp => this
        do while (associated(tmp%next))
            if (getssn(tmp%file)==s) then
                tmp%file = tmp%next%file
                tmp%next => tmp%next%next
                return
            endif
            tmp => tmp%next
        enddo
        Print *, 'Database::remove: file not found'
        end subroutine remove

!

        type (poly_Personnel) function locate(this,s)
! Find a file in the database
        type (Database), target, intent (in) :: this
        integer, intent (in) :: s
        type (Database), pointer :: tmp
        tmp => this
        do while (associated(tmp%next))
            if (getssn(tmp%file)==s) then
                locate = tmp%file
                return
            endif
            tmp => tmp%next
        enddo
        Print *, 'Database::locate: file not found'
        call new(locate)
        end function locate
end module Database_class

```

## C++ version

```
// *****  
// **** database.h ****  
// *****  
  
class Database {  
    Personnel *file;  
    Database *next;  
public:  
    Database();  
    void print();  
    void add(Personnel *pf);  
    Personnel *locate(const int ssn);  
    void remove(const int ssn);  
};
```

```

//*****
//****  database.cc  ****
//*****

#include <stream.h>
#include <string.h>

#include "personnel.h"
#include "database.h"

// Constructor
Database::Database()
{
    file=NULL;
    next=NULL;
}

// Print the database
void Database::print()
{
    Database *tmp=this;

    while (tmp->next != NULL) {
        tmp->file->print();
        tmp = tmp->next;
    }
}

// Add a file to the database
void Database::add(Personnel *f)
{
    Database *tmp=this;

    while (tmp->next != NULL) {
        tmp = tmp->next;
    }
    tmp->file = f;
    tmp->next = new Database;
}

```

```

// Remove a file from the database
void Database::remove(const int s)
{
    Database *tmp=this;

    while (tmp->next != NULL) {
        if (tmp->file->getssn() == s) {
            tmp->file = tmp->next->file;
            tmp->next = tmp->next->next;
            return;
        }
        tmp = tmp->next;
    }
    cout << "Database::remove: file not found" << endl;
}

// Find a file in the database
Personnel *Database::locate(const int s)
{
    Database *tmp=this;

    while (tmp->next != NULL) {
        if (tmp->file->getssn() == s) {
            return tmp->file;
        }
        tmp = tmp->next;
    }
    cout << "Database::locate: file not found" << endl;
    return NULL;
}

```

## Appendix F: Database Application Test Program

C++ version

```
// *****
// ****  database_test.cc  ****
// *****

#include "personnel.h"
#include "teacher.h"
#include "student.h"
#include "database.h"

main()
{
    Database cs;
    Personnel *person;
    int i;

    // Add a student file
    cs.add(new Student(1, "PAUL", "JONES"));

    // Add a teacher file
    cs.add(new Teacher(2, "JOHN", "WHITE", 1000));

    // Locate item in the database with ssn = 1
    person = cs.locate(1);

    // Add a physics class
    ((Student*)person)->addclass("PHYSICS");

    // Locate item in the database with ssn = 2
    person = cs.locate(2);

    // Update the salary
    ((Teacher*)person)->updatesalary(2000);

    // Print the database
    cs.print();

    // Delete each data file from database
    for (i=0; i < Personnel::get_num_files(); i++ ) {
        cs.remove(i+1);
    }
}
```