

## Chapter 6

# Inheritance and Polymorphism

### 6.1 Introduction

As we have seen earlier in our introduction to OOP *inheritance* is a mechanism for deriving a new class from an older *base class*. That is, the base class, sometimes called the *super class*, is supplemented or selectively altered to create the new *derived class*. Inheritance provides a powerful code reuse mechanism since a hierarchy of related classes can be created that share the same code. A class can be derived from an existing base class using the module construct illustrated in Fig. 6.1.

We note that the inheritance is invoked by the USE statement. Sometimes an inherited entity (attribute or member) needs to be slightly amended for the purposes of the new classes. Thus, at times one may want to selectively bring into the new class only certain entities from the base class. The modifier ONLY in a USE statement allows one to select the desired entities from the base class as illustrated below in Fig. 6.2. It is also common to develop name conflicts when combining entities from one or more related classes. Thus a rename modifier, =>, is also provided for a USE statement to allow the programmer to pick a new *local* name for an entity onherited from the base class. The form for that modifier is given in Fig. 6.3.

It is logical to extend any or all of the above inheritance mechanisms to produce multiple inheritance. *Multiple Inheritance* allows a derived class to be created by using inheritance from more than a single base class. While multiple inheritance may at first seem like a panacea for efficient code reuse, experience has shown that a heavy use of multiple inheritance can result in entity conflicts and be otherwise counterproductive. Nevertheless it is a useful tool in OOP. In F90 the module form for selective multiple inheritance would combine the above USE options in a single module as illustrated in Fig. 6.4.

---

```
module derived_class_name
  use base_class_name
  ! new attribute declarations, if any
  ...
contains
  ! new member definitions
  ...
end module derived_class_name
```

**Figure 6.1:** F90 Single Inheritance Form.

```

module derived_class_name
    use base_class_name, only: list_of_entities
    ! new attribute declarations, if any
        ...
contains

    ! new member definitions
        ...
end module derived_class_name

```

**Figure 6.2:** F90 Selective Single Inheritance Form.

```

module derived_class_name
    use base_class_name, local_name => base_entity_name
    ! new attribute declarations, if any
        ...
contains

    ! new member definitions
        ...
end module derived_class_name

```

**Figure 6.3:** F90 Single Inheritance Form, with Local Renaming.

```

module derived_class_name
    use base1_class_name
    use base2_class_name
    use base3_class_name, only: list_of_entities
    use base4_class_name, local_name => base_entity_name
    ! new attribute declarations, if any
        ...
contains

    ! new member definitions
        ...
end module derived_class_name

```

**Figure 6.4:** F90 Multiple Selective Inheritance with Renaming.

```

[ 1] module class_Professor      ! file: class_Professor.f90
[ 2]   implicit none
[ 3]   public :: print, name
[ 4]   private :: publs
[ 5]   type Professor
[ 6]     character (len=20) :: name
[ 7]     integer           :: publs ! publications
[ 8]   end type Professor
[ 9] contains
[10]   function make_Professor (n, p) result (who)
[11]     character (len=*), optional, intent(in) :: n
[12]     integer, optional, intent(in) :: p
[13]     type (Professor) :: who ! out
[14]     who%name         = " " ! set defaults
[15]     who%publs        = 0.0
[16]     if ( present(n) ) who%name = n ! construct
[17]     if ( present(p) ) who%publs = p
[18]   end function make_Professor
[19]
[20]   function print (who)
[21]     type (Professor), intent(in) :: who
[22]     print *, "My name is ", who%name, &
[23]           ", and I have ", who%publs, " publications."
[24]   end function print
[25] end module class_Professor

```

Figure 6.5: A Professor Class

## 6.2 Example Applications of Inheritance

### 6.2.1 The Professor Class

In the introductory examples of OOP in Chapter 3 we introduced the concepts of inheritance and multiple inheritance by the use of the `Date` class, `Person` class, and `Student` class. To reinforce those concepts we will reuse those three classes and will have them be inherited by a `Professor` class. Acknowledging the common “publish or perish” aspect of academic life the professor class must keep up with the number of publications of the professor. The new class is given in Fig. 6.5 along with a small validation program in Fig. 6.6.

Note that the validation program brings in three different versions of the “print” member (lines 7-9) and renames two of them to allow a polymorphic print statement (lines 12-14) that selects the proper member based solely on the class of its argument. Observe that the previous `Date` class is brought into the main through the use of the `Person` class (in line 7). Of course, it is necessary to have an interface defined for the overloaded member name so that the compiler knows which candidate routines to search at run time. This example also serves to remind the reader that Fortran does *not* have *keywords* that are not allowed to be used by the programmer. In this case the print function (lines 19, 22, 25) has automatically replaced the intrinsic print function of Fortran. Most languages, including C++ do not allow one to do that.

### 6.2.2 The Employee and Manager Classes

Next we will begin the popular employee-manager classes as examples of common related classes that demonstrate the use of inheritance. Once again the idea behind encapsulating these data and their associated functionality is to model a pair of real world entities - an employee and a manager. As we go through possible relations between these two simple classes it becomes clear that there is no unique way to establish the classes and how they should interact. We begin with a minimal approach and then work through two alternate versions to reach the point where an experienced OO programmer might have begun. The first `Employee` class, shown in Fig. 6.7 has a name and pay rate as its attributes. Only the intrinsic constructor is used within the member `setDataE` to concatenate a first name and last name to form the complete name attribute and to accept the pay rate. To query members `getNameE` and `getRate` are provided to extract either of the desired attributes. Finally, member `payE` is provided to compute the pay earned by an employee. It assumes that an employee is paid by the hour. A simple testing main program is shown in Fig. 6.8 It simply defines two employees (`emp11` and `emp12`), assigns their names and pay rates, and then computes and displays their pay based on the respective number of hours worked.

```

[ 1] ! Multiple Inheritance and Polymorphism of the "print" function
[ 2] include 'class_Person.inc'           ! also brings in class_Date
[ 3] include 'class_Student.inc'
[ 4] include 'class_Professor.inc'
[ 5]
[ 6] program main
[ 7]   use class_Person                     ! no changes
[ 8]   use class_Student, print_S => print ! renamed to print_S
[ 9]   use class_Professor, print_F => print ! renamed to print_F
[10]   implicit none
[11]
[12] ! Interface to generic routine, print, for any type argument
[13] interface print ! using renamed type dependent functions
[14]   module procedure print_Name, print_S, print_F
[15] end interface
[16]
[17] type (Person) :: x; type (Student) :: y; type (Professor) :: z
[18]
[19]   x = Person ("Bob"); ! default constructor
[20]   call print(x);     ! print person type
[21]
[22]   y = Student ("Tom", 3.47); ! default constructor
[23]   call print(y);     ! print student type
[24]
[25]   z = Professor ("Ann", 7); ! default constructor
[26]   call print(z);     ! print professor type
[27]   ! alternate constructors not used
[28] end program main      ! Running gives:
[29] ! Bob
[30] ! My name is Tom, and my G.P.A. is 3.4700000.
[31] ! My name is Ann, and I have 7 publications.

```

**Figure 6.6:** Bringing Four Classes and Three Functions Together

```

[ 1] module class_Employee
[ 2] ! The module class_Employee contains both the
[ 3] ! data and functionality of an employee.
[ 4] !
[ 5]   implicit none
[ 6]   public :: setDataE, getNameE, payE ! the Functionality
[ 7]
[ 8]   type Employee           ! the Data
[ 9]   private
[10]   character(30) :: name
[11]   real          :: payRate ; end type Employee
[12]
[13] contains ! inherited internal variables and subprograms
[14]
[15] function setDataE (lastName, firstName, newPayRate) result (E)
[16]   character(*), intent(in) :: lastName
[17]   character(*), intent(in) :: firstName
[18]   real,          intent(in) :: newPayRate
[19]   type (Employee)  :: E           ! employee
[20]   ! use intrinsic constructor
[21]   E = Employee(trim(firstName)//" "/trim(lastName)),newPayRate)
[22] end function setDataE
[23]
[24] function getNameE ( Person ) result ( n )
[25]   type (Employee), intent(in) :: Person
[26]   character(30)              :: n           ! name
[27]   n = Person % name ; end function getNameE
[28]
[29] function getRate ( Person ) result ( r )
[30]   type (Employee), intent(in) :: Person
[31]   real                      :: r           ! rate
[32]   r = Person % payRate ; end function getRate
[33]
[34] function payE ( Person, hoursWorked ) result ( amount )
[35]   type (Employee), intent(in) :: Person
[36]   real,                  intent(in) :: hoursWorked
[37]   real                    :: amount
[38]   amount = Person % payRate * hoursWorked ; end function payE
[39] end module class_Employee

```

**Figure 6.7:** First Definition of an Employee Class

Note that both `empl1` and `empl2` are each an instance of a class, and therefore they are objects and thus distinctly different from a class.

```

[ 1]  program main
[ 2]  ! Example use of employees
[ 3]  use class_Employee
[ 4]  type (Employee)  empl1, empl2
[ 5]
[ 6]  ! Set up 1st employee and print out his name and pay
[ 7]  empl1 = setDataE ( "Jones", "Bill", 25.0 )
[ 8]  print *, "Name: ", getNameE ( empl1 )
[ 9]  print *, "Pay: ", payE ( empl1, 40.0 )
[10]
[11] ! Set up 2nd employee and print out her name and pay
[12] empl2 = setDataE ( "Doe", "Jane", 27.5 )
[13] print *, "Name: ", getNameE ( empl2 )
[14] print *, "Pay: ", payE ( empl2, 38.0 )
[15] end program main      ! Running produces:
[16] ! Name: Bill Jones    ! Pay:    1000.
[17] ! Name: Jane Doe     ! Pay:    1045.

```

**Figure 6.8:** First Test of an Employee Class

Next we deal with a manager which Is-A “kind of” employee. One difference is that some managers may be paid a salary rather than an hourly rate. Thus we have the `Manager` class inherit the attributes of the `Employee` class and add a new logical attribute `isSalaried` which is true when the manager is salary based. To support such a case we must add a new member `setSalaried` which can turn the new attribute on or off, and a corresponding member `payM` that uses the `isSalaried` flag when computing the pay. The `class_Manager` module is shown in Fig. 6.9 Note that the constructor `Manager_` defaults to an hourly worker (line 33) and it uses the inherited employee constructor (line 31). Figure 6.10 shows a test program to validate the manager class (and indirectly the employee class). It defines a salaried manager, `mgr1`, an hourly manager `mgr2`, and prints the name and weekly pay for both. (Verify these weekly pay amounts.)

With these two classes we have mainly used different program names for members that do similar things in each class (the author’s preference). However, many programmers prefer to use a single member name for a typical operation, regardless of the class of the operand. We also restricted all the attributes to `private` and allowed all the members to be `public`. We could use several alternate approaches to building our `Employee` and `Manager` classes. For example, assume we want a single member name called `pay` to be invoked for an employee, or manager (or executive). Furthermore we will allow the attributes to be `public` instead of `private`. Lowering the access restrictions to the attributes makes it easier to write an alternate program, but it is not a recommended procedure since it breaks the data hiding concept that has been shown to be important to OO software maintenance and reliability. The alternate `Employee` and `Manager` classes are shown in Figs. 6.11 and 6.12, respectively. Note that they both have a `pay` member but their arguments are of different classes and their internal calculations are different. Now we want a validation program that will create both classes of individuals, and use a single member name, `PrintPay`, to print the proper pay amount from the single member name `pay`. This can be done in different ways. One problem that arises in our plan to reuse the code in the two alternate class modules is that neither contained a pay printing member. We will need two new routines, `PrintPayEmployee` and `PrintPayManager`, and a generic or polymorphic interface to them. We have at least three ways to do this. One way is to place the two routines in an external file (or external to main if in the same file), leave the two class modules unchanged, and have the `main` program begin with (or `INCLUDE`) an external interface prototype. This first approach to `main` is shown in Fig. 6.13. Note that the two new external routines must each use their respective class module.

A second approach would be to have the two new routines become internal to the `main`, after line 30, and occur before `end program`. Another change would be that each routine would have to omit its `use` statement (such as lines 34 and 41). Why? Because they are now internal to `main` and it has already made use of the two classes (in line 2). That approach is shown in Figs. 6.13

A third approach would be the most logical and consistent with OOP principles. It is to make all the class attributes `private`, place the print members in each respective class, insert a single generic name interface in each class, and modify the `main` program to use the polymorphic name regardless of the class of the argument it acts upon. The improved version of the classes are given below in Figs. 6.14, 6.15, and 6.16. Observe that generic interfaces for `PrintPay` and `getName` have been added, but that we could

```

[ 1] module class_Manager
[ 2] ! Gets class_Employee and add additional functionality
[ 3] use class_Employee
[ 4] implicit none
[ 5] public :: setSalaried, payM
[ 6]
[ 7] type Manager          ! the Data
[ 8] private
[ 9] type (Employee) :: Person
[10] integer          :: isSalaried ! ( or logical )
[11] end type Manager
[12]
[13] contains ! inherited internal variables and subprograms
[14]
[15] function getEmployee ( M ) result (E)
[16] type (Manager ), intent(in) :: M
[17] type (Employee)           :: E
[18] E = M % Person ; end function getEmployee
[19]
[20] function getNameM ( M ) result (n)
[21] type (Manager ), intent(in) :: M
[22] type (Employee)           :: E
[23] character(30)             :: n          ! name
[24] n = getNameE(M % Person); end function getNameM
[25]
[26] function Manager_ (lastName, firstName, newPayRate) result (M)
[27] character(*), intent(in) :: lastName
[28] character(*), intent(in) :: firstName
[29] real,          intent(in) :: newPayRate
[30] type (Employee)           :: E          ! employee
[31] type (Manager )          :: M          ! manager constructor
[32] E = setDataE (lastName, firstName, newPayRate)
[33] ! use intrinsic constructor
[34] M = Manager(E, 0) ! default to no salary
[35] end function Manager_
[36]
[37] function setDataM (lastName, firstName, newPayRate) result (M)
[38] character(*), intent(in) :: lastName
[39] character(*), intent(in) :: firstName
[40] real,          intent(in) :: newPayRate
[41] type (Employee)           :: E          ! employee
[42] type (Manager )          :: M          ! manager
[43] E = setDataE (lastName, firstName, newPayRate)
[44] M % Person = E
[45] end function setDataM
[46]
[47] subroutine setSalaried ( Who, salariedFlag )
[48] type (Manager), intent(inout) :: Who
[49] integer,          intent(in)   :: salariedFlag
[50] Who % isSalaried = salariedFlag ; end subroutine setSalaried
[51]
[52] function payM ( Human, hoursWorked ) result ( amount )
[53] type (Manager), intent(in) :: Human
[54] real,          intent(in) :: hoursWorked
[55] real,          :: amount, value
[56] value = getRate( getEmployee(Human) )
[57] if ( Human % isSalaried == 1 ) then ! (or use logical)
[58] amount = value
[59] else
[60] amount = value * hoursWorked
[61] end if ; end function payM
[62] end module class_Manager

```

**Figure 6.9:** A First Declaration of a Manager Class

not do that for a corresponding setData; do you know why? A final improvement will be given as an assignment.

## 6.3 Polymorphism

Fortran 90 and 95 do not include the full range of polymorphism abilities that one would like to have in an object-oriented language. It is expected that the Fortran 2000 standard will add those abilities.

Some of the code “re-use” features can be constructed through the concept of subprogram “templates,” which will be discussed below. The lack of a standard “Is\_A” polymorphism can be overcome in F90/95 by the use of the SELECT CASE feature to define “sub-types” of objects. This approach of sub-typing programming provides the desired additional functionality, but it is clearly not as easy to change or extend as an inheritance feature built into the language standard. A short example will be provided.

```

[ 1] program main ! Example use of managers
[ 2]   use class_Manager
[ 3]   implicit none
[ 4]   type (Manager) mgr1, mgr2
[ 5]
[ 6]   ! Set up 1st manager and print out her name and pay
[ 7]
[ 8]   mgr1 = setDataM ( "Smith", "Kimberly", 1900.0 )
[ 9]   call setSalaried ( mgr1, 1 ) ! Has a salary
[10]
[11]   print *, "Name: ", getNameM ( mgr1)
[12]   print *, "Pay: ", payM ( mgr1, 40.0 )
[13]
[14]   ! Set up 2nd manager and print out his name and pay
[15]
[16]   ! mgr2 = setDataM ( "Danish", "Tom", 46.5 )
[17]   ! call setSalaried ( mgr2, 0 ) ! Doesn't have a salary
[18]   ! or
[19]   mgr2 = Manager_ ( "Danish", "Tom", 46.5 )
[20]
[21]   print *, "Name: ", getNameM ( mgr2)
[22]   print *, "Pay: ", payM ( mgr2, 40.0 )
[23] end program main ! Running produces:
[24] ! Name: Kimberly Smith ! Pay: 1900.
[25] ! Name: Tom Danish ! Pay: 1860.

```

**Figure 6.10:** First Test of a Manager Class

```

[ 1] module class_Employee ! Alternate
[ 2]   implicit none
[ 3]   public :: setData, getName, pay ! the Functionality
[ 4]
[ 5]   type Employee ! the Data
[ 6]     character(30) :: name
[ 7]     real :: payRate
[ 8]   end type Employee
[ 9]
[10] contains ! inherited internal variables and subprograms
[11]
[12]   subroutine setData ( Person, lastName, firstName, newPayRate )
[13]     type (Employee) :: Person
[14]     character(*) :: lastName
[15]     character(*) :: firstName
[16]     real :: newPayRate
[17]     Person % name = trim (firstName) // " " // trim (lastName)
[18]     Person % payRate = newPayRate
[19]   end subroutine setData
[20]
[21]   function getName ( Person )
[22]     character(30) :: getName
[23]     type (Employee) :: Person
[24]     getName = Person % name
[25]   end function getName
[26]
[27]   function pay ( Person, hoursWorked )
[28]     real :: pay
[29]     type (Employee) :: Person
[30]     real :: hoursWorked
[31]     pay = Person % payRate * hoursWorked
[32]   end function pay
[33] end module class_Employee

```

**Figure 6.11:** Alternate Public Access Form of an Employee Class

### 6.3.1 Templates

One of our goals has been to develop software that can be reused for other applications. There are some algorithms that are effectively independent of the object type on which they operate. For example, in a sorting algorithm one often needs to interchange, or swap, two objects. A short routine for that purpose follows:

```

subroutine swap_integers (x, y)
  implicit none
  integer, intent(inout) :: x, y
  integer :: temp
  temp = x
  x = y
  y = temp
end subroutine swap_integers

```

```

[ 1] module class_Manager ! Alternate
[ 2]   use class_Employee, payEmployee => pay ! renamed
[ 3]   implicit none
[ 4]   public :: setSalaried, payManager
[ 5]
[ 6]   type Manager ! the Data
[ 7]     type (Employee) :: Person
[ 8]     integer :: isSalaried ! ( or logical )
[ 9]   end type Manager
[10]
[11] contains ! inherited internal variables and subprograms
[12]
[13]   subroutine setSalaried ( Who, salariedFlag )
[14]     type (Manager) :: Who
[15]     integer :: salariedFlag
[16]     Who % isSalaried = salariedFlag
[17]   end subroutine setSalaried
[18]
[19]   function pay ( Human, hoursWorked )
[20]     real :: pay
[21]     type (Manager) :: Human
[22]     real :: hoursWorked
[23]
[24]     if ( Human % isSalaried == 1 ) then ! (or use logical)
[25]       pay = Human % Person % payRate
[26]     else
[27]       pay = Human % Person % payRate * hoursWorked
[28]     end if
[29]   end function pay
[30] end module class_Manager

```

**Figure 6.12:** Alternate Public Access Form of a Manager Class

Observe that in this form it appears necessary to have one version for integer arguments and another for real arguments. Indeed we might need a different version of the routine for each type of argument that you may need to swap. A slightly different approach would be to write our swap algorithm as:

```

subroutine swap_objects (x, y)
  implicit none
  type (Object), intent(inout) :: x, y
  type (Object) :: temp
  temp = x
  x = y
  y = temp
end subroutine swap_objects

```

which would be a single routine that would work for any Object, but it has the disadvantage that one find a way to redefine the Object type for each application of the routine. That would not be an easy task. (While we will continue with this example with the algorithm in the above forms it should be noted that the above approaches would not be efficient if x and y were very large arrays or derived type objects. In that case we would modify the algorithm slightly to employ pointers to the large data items and simply swap the pointers for a significant increase in efficiency.)

Consider ways that we might be able to generalize the above routines so that they could accept and swap any specific type of arguments. For example, the first two versions could be re-written in a so called template form as:

```

subroutine swap_Template$(x, y)
  implicit none
  Template$, intent(inout) :: x, y
  Template$ :: temp
  temp = x
  x = y
  y = temp
end subroutine swap_Template$

```

In the above template the dollar sign (\$) was included in the “wild card” because while it is a valid member of the F90 character set it is not a valid character for inclusion in the name of a variable, derived type, function, module, or subroutine. In other words, a template in the illustrated form would not compile, but such a name could serve as a reminder that its purpose is to produce a code that can be compiled after the “wild card” substitutions have been made.

With this type of template it would be very easy to use a modern text editor to do a global substitution of any one of the intrinsic types character, complex, double precision, integer, logical, or real for the “wild card” keyword Template\$ to produce a source code to swap any or all of

```

[ 1] program main ! Alternate employee and manager classes
[ 2]   use class_Manager ! and thus Employee
[ 3]   implicit none
[ 4]   ! supply interface for external code not in classes
[ 5]   interface PrintPay ! For TYPE dependent arguments
[ 6]     subroutine PrintPayManager ( Human, hoursWorked )
[ 7]       use class_Manager
[ 8]       type (Manager) :: Human
[ 9]       real            :: hoursWorked
[10]     end subroutine
[11]     subroutine PrintPayEmployee ( Person, hoursWorked )
[12]       use class_Employee
[13]       type (Employee) :: Person
[14]       real            :: hoursWorked
[15]     end subroutine
[16]   end interface
[17]
[18]   type (Employee) empl ; type (Manager) mgr
[19]
[20]   ! Set up an employee and print out his name and pay
[21]   call setData ( empl, "Burke", "John", 25.0 )
[22]
[23]   print *, "Name: ", getName ( empl )
[24]   call PrintPay ( empl, 40.0 )
[25]
[26]   ! Set up a manager and print out her name and pay
[27]   call setData ( mgr % Person, "Kovacs", "Jan", 1200.0 )
[28]   call setSalaried ( mgr, 1 ) ! Has a salary
[29]
[30]   print *, "Name: ", getName ( mgr % Person )
[31]   call PrintPay ( mgr, 40.0 )
[32] end program
[33]
[34] subroutine PrintPayEmployee ( Person, hoursWorked )
[35]   use class_Employee
[36]   type (Employee) :: Person
[37]   real            :: hoursWorked
[38]   print *, "Pay: ", pay ( Person, hoursworked )
[39] end subroutine
[40]
[41] subroutine PrintPayManager ( Human, hoursWorked )
[42]   use class_Manager
[43]   type (Manager) :: Human
[44]   real            :: hoursWorked
[45]   print *, "Pay: ", pay ( Human , hoursworked )
[46] end subroutine
[47] ! Running produces;
[48] ! Name: John Burke
[49] ! Pay: 1000.
[50] ! Name: Jan Kovacs
[51] ! Pay: 1200.

```

**Figure 6.13:** Testing the Alternate Employee and Manager Classes

the intrinsic data types. There would be no need to keep up with all the different routine names if we placed all of them in a single module and also created a generic interface to them such as:

```

module swap_library
  implicit none
  interface swap ! the generic name
    module procedure swap_character, swap_complex
    module procedure swap_double_precision, swap_integer
    module procedure swap_logical, swap_real
  end interface
  contains
    subroutine swap_characters (x, y)
    .
    .
    .
  end subroutine swap_characters
  subroutine swap_ . . .
  .
  .
end module swap_library

```

The use of a text editor to make such substitutions is not very elegant and we expect that there may be a better way to pursue the concept of developing a reusable software template. The concept of a text editor substitution also fails when we go to the next logical step and try to use a derived type argument instead of any of the intrinsic data types. For example, if we were to replace the “wild card” with our previous type (chemical\_element) that would create:

```

subroutine swap_type (chemical_element) (x,y)
  implicit none

```

```

[ 1] module class_Employee                ! the base class
[ 2]   implicit none                    ! strong typing
[ 3]   private :: PrintPayEmployee, payE ! private members
[ 4]   type Employee                    ! the Data
[ 5]     private                        ! all attributes private
[ 6]     character(30) :: name
[ 7]     real          :: payRate ; end type Employee
[ 8]
[ 9]   interface PrintPay                ! a polymorphic member
[10]     module procedure PrintPayEmployee ; end interface
[11]   interface getName                 ! a polymorphic member
[12]     module procedure getNameE      ; end interface
[13] ! NOTE: can not have polymorphic setData. Why ?
[14]
[15] contains ! inherited internal variables and subprograms
[16]
[17]   function setDataE (lastName, firstName, newPayRate) result (E)
[18]     character(*), intent(in) :: lastName
[19]     character(*), intent(in) :: firstName
[20]     real,          intent(in) :: newPayRate ! amount per period
[21]     type (Employee) :: E ! employee
[22]     ! use intrinsic constructor
[23]     E = Employee(trim(firstName)//" " //trim(lastName)),newPayRate)
[24]   end function setDataE
[25]
[26]   function getNameE ( Person ) result (n)
[27]     type (Employee), intent(in) :: Person
[28]     character(30)                :: n ! name
[29]     n = Person % name ; end function getNameE
[30]
[31]   function getRate ( Person ) result ( r )
[32]     type (Employee), intent(in) :: Person
[33]     real                        :: r ! rate of pay
[34]     r = Person % payRate ; end function getRate
[35]
[36]   function payE ( Person, hoursWorked ) result ( amount )
[37]     type (Employee), intent(in) :: Person
[38]     real,          intent(in) :: hoursWorked
[39]     real                        :: amount
[40]     amount = Person % payRate * hoursWorked ; end function payE
[41]
[42]   subroutine PrintPayEmployee ( Person, hoursWorked )
[43]     type (Employee) :: Person
[44]     real            :: hoursWorked
[45]     print *, "Pay: ", payE ( Person, hoursWorked )
[46]   end subroutine
[47] end module class_Employee

```

**Figure 6.14:** A Better Private Access Form of an Employee Class

```

[ 1] module class_Manager                ! the derived class
[ 2] ! Get class_Employee, add additional attribute & members
[ 3] use class_Employee                ! inherited base class
[ 4] implicit none                    ! strong typing
[ 5] private :: PrintPayManager, payM, getNameM ! private members
[ 6]
[ 7]   type Manager                    ! the Data
[ 8]     private                        ! all attributes private
[ 9]     type (Employee) :: Person
[10]     integer          :: isSalaried ! 1 if true (or use logical)
[11]   end type Manager
[12]
[13]   interface PrintPay                ! a polymorphic member
[14]     module procedure PrintPayManager ; end interface
[15]   interface getName                 ! a polymorphic member
[16]     module procedure getNameM      ; end interface

```

*Fig. 6.15: A Better Private Access Form of a Manager Class (continued)*

```

      type (chemical_element), intent (inout)::x,y
      type (chemical_element)      ::temp
      temp = x
      x    = y
      y    = temp
    end subroutine swap_type (chemical_element)

```

This would fail to compile because it violates the syntax for a valid function or subroutine name, as well as the end function or end subroutine syntax. Except for the first and last line syntax errors this would be a valid code. To correct the problem we simply need to add a little logic and omit the characters `type`

```

[17]
[18] contains ! inherited internal variables and subprograms
[19]
[20] function getEmployee ( M ) result ( E )
[21]   type (Manager ), intent(in) :: M
[22]   type (Employee)      :: E
[23]   E = M % Person ; end function getEmployee
[24]
[25] function getNameM ( M ) result ( n )
[26]   type (Manager ), intent(in) :: M
[27]   type (Employee)      :: E
[28]   character(30)        :: n          ! name
[29]   n = getNameE(M % Person); end function getNameM
[30]
[31] function Manager_ (lastName, firstName, newPayRate) result ( M )
[32]   character(*), intent(in) :: lastName
[33]   character(*), intent(in) :: firstName
[34]   real,          intent(in) :: newPayRate
[35]   type (Employee)      :: E          ! employee
[36]   type (Manager )     :: M          ! manager constructed
[37]   E = setDataE (lastName, firstName, newPayRate)
[38]                                     ! use intrinsic constructor
[39]   M = Manager(E, 0)                  ! default to hourly
[40] end function Manager_
[41]
[42] function setDataM (lastName, firstName, newPayRate) result ( M )
[43]   character(*), intent(in) :: lastName
[44]   character(*), intent(in) :: firstName
[45]   real,          intent(in) :: newPayRate      ! hourly OR weekly
[46]   type (Employee)      :: E          ! employee
[47]   type (Manager )     :: M          ! manager constructed
[48]   E = setDataE (lastName, firstName, newPayRate)
[49]   M % Person = E ; M % isSalaried = 0      ! default to hourly
[50] end function setDataM
[51]
[52] subroutine setSalaried ( Who, salariedFlag ) ! 0=hourly, 1=weekly
[53]   type (Manager), intent(inout) :: Who
[54]   integer,          intent(in)   :: salariedFlag ! 0 OR 1
[55]   Who % isSalaried = salariedFlag ; end subroutine setSalaried
[56]
[57] function payM ( Human, hoursWorked ) result ( amount )
[58]   type (Manager), intent(in) :: Human
[59]   real,          intent(in) :: hoursWorked
[60]   real,          :: amount, value
[61]   value = getRate( getEmployee(Human) )
[62]   if ( Human % isSalaried == 1 ) then
[63]     amount = value          ! for weekly person
[64]   else
[65]     amount = value * hoursWorked ! for hourly person
[66]   end if ; end function payM
[67]
[68] subroutine PrintPayManager ( Human, hoursWorked )
[69]   type (Manager) :: Human
[70]   real           :: hoursWorked
[71]   print *, "Pay: ", payM ( Human , hoursworked )
[72] end subroutine
[73] end module class_Manager

```

**Figure 6.15:** A Better Private Access Form of a Manager Class

( ) when we create a function, module, or subroutine name that is based on a derived type data entity. Then we obtain

```

subroutine swap_chemical_element (x,y)
  implicit none
  type (chemical_element), intent (inout)::x,y
  type (chemical_element)                ::temp
  temp = x
  x = y
  y = temp
end subroutine swap_chemical_element

```

which yields a completely valid routine.

Unfortunately, text editors do not offer us such logic capabilities. However, as we have seen, high level programming languages like C++ and F90 do have those abilities. At this point you should be able to envision writing a pre-processor program that would accept a file of template routines, replace the template “wildcard” words with the desired generic forms to produce a module or header file containing the expanded source files that can then be brought into the desired program with an include or use statement. The C++ language includes a template pre-processor to expand template files as needed.

```

[ 1] program main ! Final employee and manager classes
[ 2]   use class_Manager ! and thus class_Employee
[ 3]   implicit none
[ 4]
[ 5]   type (Employee) empl ; type (Manager) mgr
[ 6]
[ 7]   ! Set up a hourly employee and print out his name and pay
[ 8]   empl = setDataE ( "Burke", "John", 25.0 )
[ 9]
[10]   print *, "Name: ", getName ( empl )
[11]   call PrintPay ( empl, 40.0 )      ! polymorphic
[12]
[13]   ! Set up a weekly manager and print out her name and pay
[14]   mgr = setDataM ( "Kovacs", "Jan", 1200.0 )
[15]   call setSalaried ( mgr, 1 )     ! rate is weekly
[16]
[17]   print *, "Name: ", getName ( mgr )
[18]   call PrintPay ( mgr, 40.0 )     ! polymorphic
[19] end program                       ! Running produces:
[20] ! Name: John Burke
[21] ! Pay:    1000.
[22] ! Name: Jan Kovacs
[23] ! Pay:    1200.

```

**Figure 6.16:** Testing the Better Employee-Manager Forms

Some programmers criticize F90/95 for not offering this ability as part of the standard. A few C++ programmers criticize templates and advise against their use. Regardless of the merits of including template pre-processors in a language standard it should be clear that it is desirable to plan software for its efficient reuse.

With F90 if one wants to take advantage of the concepts of templates then the only choices are to carry out a little text editing or develop a pre-processor with the outlined capabilities. The former is clearly the simplest and for many projects may take less time than developing such a template pre-processor. However, if one makes the time investment to produce a template pre-processor one would have a tool that could be applied to basically any coding project.

### 6.3.2 Subtyping Objects (Dynamic Dispatching)

One polymorphic feature missing from the Fortran 90 standard (but expected in Fortran 2000) that is common to most object oriented languages is called run-time polymorphism or dynamic dispatching. In the C++ language this ability is introduced in the so-called “virtual function.” To emulate this ability is quite straightforward in F90 but is not elegant since it usually requires a group of if-elseif statements or other selection processes. It is only tedious if the inheritance hierarchy contains many unmodified subroutines and functions. The importance of the lack of a standardized dynamic dispatching depends on the problem domain to which it must be applied. For several applications demonstrated in the literature the alternate use of subtyping has worked quite well and resulted in programs that have been shown to run several times faster than equivalent C++ versions.

We implement dynamic dispatching in F90 by a process often called subtyping. Two features must be constructed to do this. First, a pointer object, which can point to any subtype member in an inheritance hierarchy, must be developed. Second, an if-elseif or other selection method is developed to serve as a dispatch mechanism to select the unique appropriate procedure to be executed based on the actual class referenced in the controlling pointer object. This subtyping process is also referred to as implementing a polymorphic class. Of course, the details of the actual dispatching process can be hidden from the procedures that utilize the polymorphic class.

This process will be illustrated by creating a specific polymorphic class, called `Is_A_Member_Class`, which has polymorphic procedures named `new`, `assign`, and `display`. They will construct a new instance of the object, assign it a value, and list its components. The minimum example of such a process requires two members and is easily extended to any number of member classes. We begin by defining each of the subtype classes of interest.

The first is a class, `Member_1_Class`, which has two real components and the encapsulated functionality to construct a new instance and another to accept a pointer to such a subtype and display related information. It is shown in Fig. 6.17. The next class, `Member_2_Class`, has three components: two reals and one of type `Member_1`. It has the same sort of functionality, but clearly must act on more

```

[ 1] Module Member_1_Class
[ 2]   implicit none
[ 3]   type member_1
[ 4]     real :: real_1, real_2
[ 5]   end type member_1
[ 6]
[ 7] contains
[ 8]
[ 9]   subroutine new_member_1 (member, a, b)
[10]     real, intent(in) :: a, b
[11]     type (member_1) :: member
[12]     member%real_1 = a ; member%real_2 = b
[13]   end subroutine new_member_1
[14]
[15]   subroutine display_memb_1 (pt_to_memb_1, c)
[16]     type (member_1), pointer :: pt_to_memb_1
[17]     character(len=1), intent(in) :: c
[18]     print *, 'display_memb_1 ', c
[19]   end subroutine display_memb_1
[20]
[21] End Module Member_1_Class

```

**Figure 6.17: Defining Subtype\_1**

```

[ 1] Module Member_2_Class
[ 2]   Use Member_1_Class
[ 3]   implicit none
[ 4]   type member_2
[ 5]     type (member_1) :: r_1_2
[ 6]     real :: real_3, real_4
[ 7]   end type member_2
[ 8]
[ 9] contains
[10]
[11]   subroutine new_member_2 (member, a, b, c, d)
[12]     real, intent(in) :: a, b, c, d
[13]     type (member_2) :: member
[14]     call new_member_1 (member%r_1_2, a, b)
[15]     member%real_3 = c ; member%real_4 = d
[16]   end subroutine new_member_2
[17]
[18]   subroutine display_memb_2 (pt_to_memb_2, c)
[19]     type (member_2), pointer :: pt_to_memb_2
[20]     character(len=1), intent(in) :: c
[21]     print *, 'display_memb_2 ', c
[22]   end subroutine display_memb_2
[23]
[24] End Module Member_2_Class

```

**Figure 6.18: Defining Subtype\_2**

components. It has also inherited the functionality from the Member\_1\_Class; as displayed in Fig. 6.18.

The polymorphic class is called the Is\_A\_Member\_Class and is shown in Fig. 6.19. It includes all of the encapsulated data and function's of the above two subtypes by including their use statements. The necessary pointer object is defined as an Is\_A\_Member type that has a unique pointer for each subtype member (two in this case). It also defines a polymorphic interface to each of the common procedures to be applied to the various subtype objects. In the polymorphic function assigning the dispatching is done very simply. First, all pointers to the family of subtypes are nullified, and then the unique pointer component to the subtype of interest is set to point to the desired member. The dispatching process for the display procedure is different. It requires an if-elseif construct that contains calls to all of the possible subtype members (two here) and a failsafe default state to abort the process or undertake the necessary exception handling. Since all but one of the subtype pointer objects have been nullified it employs the associated intrinsic function to select the one, and only, procedure to call and passes the pointer object on to that procedure. The validation of this dynamic dispatching through a polymorphic class is shown in Fig. 6.20. There a target is declared for each possible subtype and then each of them is constructed and sent on to the other polymorphic functions. The results clearly show that different display procedures were used depending on the class of object supplied as an argument. It is expected that the new Fortran 2000 standard will allow such dynamic dispatching in a much simpler fashion.

```

[ 1] Module Is_A_Member_Class
[ 2] Use Member_1_Class ; Use Member_2_Class
[ 3]   implicit none
[ 4]
[ 5]   type Is_A_Member
[ 6]     private
[ 7]     type (member_1), pointer :: pt_to_memb_1
[ 8]     type (member_2), pointer :: pt_to_memb_2
[ 9]   end type Is_A_Member
[10]
[11]   interface new
[12]     module procedure new_member_1
[13]     module procedure new_member_2
[14]   end interface
[15]
[16]   interface assign
[17]     module procedure assign_memb_1
[18]     module procedure assign_memb_2
[19]   end interface
[20]
[21]   interface display
[22]     module procedure display_memb_1
[23]     module procedure display_memb_2
[24]   end interface
[25]
[26] contains
[27]
[28]   subroutine assign_memb_1 (Family, member)
[29]     type (member_1), target, intent(in) :: member
[30]     type (Is_A_Member),      intent(out) :: Family
[31]     call nullify_Is_A_Member (Family)
[32]     Family%pt_to_memb_1 => member
[33]   end subroutine assign_memb_1
[34]
[35]   subroutine assign_memb_2 (Family, member)
[36]     type (member_2), target, intent(in) :: member
[37]     type (Is_A_Member),      intent(out) :: Family
[38]     call nullify_Is_A_Member (Family)
[39]     Family%pt_to_memb_2 => member
[40]   end subroutine assign_memb_2
[41]
[42]   subroutine nullify_Is_A_Member (Family)
[43]     type (Is_A_Member), intent(inout) :: Family
[44]     nullify (Family%pt_to_memb_1)
[45]     nullify (Family%pt_to_memb_2)
[46]   end subroutine nullify_Is_A_Member
[47]
[48]   subroutine display_members (A_Member, c)
[49]     type (Is_A_Member), intent(in) :: A_Member
[50]     character(len=1),   intent(in) :: c
[51]
[52]     ! select the proper member
[53]     if ( associated (A_Member%pt_to_memb_1) ) then
[54]       call display (A_Member%pt_to_memb_1, c)
[55]     else if ( associated (A_Member%pt_to_memb_2) ) then
[56]       call display (A_Member%pt_to_memb_2, c)
[57]     else ! default case
[58]       stop 'Error, no member defined in Is_A_Member_Class'
[59]     end if
[60]   end subroutine display_members
[61] End Module Is_A_Member_Class

```

**Figure 6.19:** Combining Subtypes in an Is\_A Class

```

[ 1] program main
[ 2] use Is_A_Member_Class
[ 3]   implicit none
[ 4]
[ 5]   type (Is_A_Member)      :: generic_member
[ 6]   type (member_1), target :: pt_to_memb_1
[ 7]   type (member_2), target :: pt_to_memb_2
[ 8]   character(len=1) :: c
[ 9]
[10]     c = 'A'
[11]     call new (pt_to_memb_1, 1.0, 2.0)
[12]     call assign (generic_member, pt_to_memb_1)
[13]     call display_members (generic_member, c)
[14]
[15]     c = 'B'
[16]     call new (pt_to_memb_2, 1.0, 2.0, 3.0, 4.0)
[17]     call assign (generic_member, pt_to_memb_2)
[18]     call display_members (generic_member, c)
[19]
[20] end program main
[21] ! running gives
[22] ! display_memb_1 A
[23] ! display_memb_2 B

```

**Figure 6.20:** Testing the Is\_A Subtypes

## 6.4 Exercises

1. Write a main program that will use the Class\_X and Class\_Y, given below, to invoke each of the f(v) routines and assign a value of 66 to the integer component in X, and 44 to the integer component in Y. (Solution given.)

```

module class_X
  public :: f
  type X_; integer a; end type X_
contains ! functionality
  subroutine f(v); type (X_), intent(in) :: v
    print *, "X_ f() executing"; end subroutine
end module class_X

module class_Y
  use class_X, X_f => f ! renamed
  public :: f
  type Y_; integer a; end type Y_ ! dominates X_
contains ! functionality, overrides X_ f()
  subroutine f(v); type (Y_), intent(in) :: v
    print *, "Y_ f() executing"; end subroutine
end module class_Y

```

2. Create the generic interface that would allow a single constructor name, Position\_Angle\_, to be used for all the constructors given in the previous chapter for the class Position\_Angle. Note that this is possible because they all had unique argument signatures. Also provide a new main program to test this polymorphic version.

3. Modify the last Manager class by deleting the member setDataM and replace its appearance in the last main with an existing constructor (but not used) in that class. Also provide a generic setData interface in the class Employee as a nicer name and to allow for other employees, like executives, that may have different kinds of attributes that may need to be set in the future. Explain why we could not use setDataM in the generic setData.

4. The final member setDataE in Employee is actually a constructor and the name is misleading since it does not just set data values, it also builds the name. Rename setDataE to the constructor notation Employee\_ and provide a new member in Employee called setRateE that only sets the employee pay rate.