Inheritance

**Inheritance** is a mechanism to create a hierarchy of classes in which a parent (or base) class contains the common properties of the hierarchy and child (or derived) classes can modify (or specialize) these properties. The value of inheritance is to avoid duplicating code when creating classes which are similar to one another.

In object-oriented languages, inheritance specifically refers to a data relationship where the child class contains the data and functions of the parent. This allows the child to act as a parent for those functions that the child is not modifying. In other words, an inheritance relationship is like that of a Russian matrioska doll, where one doll fits inside another.

Inheritance is automated in object-oriented languages because of this special relationship. In Fortran95, however, inheritance must be manually constructed.

To see how this works, let us continue with the Henderson and Zorn example. Last time we constructed a general class called Personnel. Now suppose we want to create a special class for Student records which contains additional information specific to students, namely the number and name of classes they are taking, but the other information for a student is the same as before, and we would like to avoid rewriting that code for the Student class. We do this by defining a Student class to explicitly contain a Personnel object in its class definition:
type Student  ! Student inherits from Personnel
type (Personnel) :: personnel
  integer :: nclasses
  character(len=12), dimension(10) :: classes
end type Student

In an inheritance relationship, all the functions of the parent class must also function in the child class. In other words, the following functions must also work:

  subroutine new_Student(this,s,fn,ln)
  subroutine delete_Student(this)
  subroutine print_Student(this)
  function getssn_Student(this) result(ssn)

To implement these functions in the child class, we delegate to the parent class part of the work. For example, the constructor for Student uses the constructor for Personnel to initialize the Personnel component of student, then adds the initialization of the classes the student is taking, as follows:

  subroutine new_Student(this,s,fn,ln)
    type (Student), intent (in) :: this
    integer, intent (in) :: s
    character(len=*)& intent (in) :: fn, ln
    call new_Personnel(this%personnel,s,fn,ln) ! delegation
    this%nclasses = 0 ! new information
    this%classes = ‘’
  end subroutine new_Student
In a similar manner, we implement the print function by delegating to the Personnel class the responsibility to print out the Personnel component of the Student record, then print out the new information relating to students:

```fortran
subroutine print_Student(this)
  type (Student), intent (in) :: this
  call print_Personnel(this%personnel) ! delegation
  print *, 'Enrolled in: ', this%nclasses, ' class(es)'
  do i = 1, this%nclasses
    print *, this%classes(i)
  enddo
end subroutine print_Student
```

In an object-oriented language, the language simplifies the implementation of an inheritance hierarchy. One simplification is that the Personnel component of Student does not have to be explicitly declared (although the components are available). The second simplification is that if the function in the Student class is the same as the function in the Personnel class, then it would be automatically implemented. In Fortran95 this must be still be done manually, for example:

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

would not have to be written in an OO language.
In order for this student class to be useful, we create a procedure to add a class to the student’s record.

```fortran
subroutine add_class(this,c)
  type (Student), intent (inout) :: this
  character(len=*), intent(in) :: c
  this%nclasses = this%nclasses + 1 ! increment counter
  this%classes(this%nclasses) = c ! add class name
end subroutine add_class
```

Let’s see how this works in a program:

```fortran
program database
  use Student_class
  type (Student) :: studentA ! declare object
  call new_Student(studentA,1,'Pat','Smith') ! Constructor
  call add_class(studentA,'Physics 290') ! add class
  call print_Student(studentA) ! print record
  ...
end program database
```

The output will be

```
1 Pat Smith
Enrolled in: 1 class(es)
Physics 290
```

Note that the child class Student does not need to know anything about how the Personnel class was implemented.
Let’s see what this Student class looks like:

module Student_class
use Personnel_class   ! Student inherits from Personnel

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

interface printout       ! overload print procedure
   module procedure print_Student
end interface

interface delete        ! overload destructor
   module procedure delete_Student
end interface

interface getssn        ! overload accessor function
   module procedure getssn_Student
end interface

contains
   subroutine new_Student(this,s,fn,ln)
      ...
   subroutine delete_Student(this)
      ...
   subroutine print_Student(this,printssn)
      ...
   function getssn_Student(this) result(ssn)
      ...
   subroutine add_class(this,c)
      ...
end module Student_class
In a similar manner to the Student class, one can also derive a Teacher class from Personnel, which contains additional information specific to teachers, namely their salary.

```fortran
  type Teacher              ! Teacher inherits from Personnel
    type (Personnel) :: personnel
    integer :: salary
  end type Teacher
```

The functions of the parent class must be extended to work in the Teacher class, as we did for the Student class:

```fortran
  subroutine new_Teacher(this,s,fn,ln)
  subroutine delete_Teacher(this)
  subroutine print_Teacher(this)
  function getssn_Teacher(this) result(ssn)
```

In addition, we provide a new procedure, to update the salary in the teacher’s record:

```fortran
  subroutine update_salary(this,sal)
    type (Teacher), intent (inout) :: this
    integer, intent(in) :: sal
    this%salary = sal                  ! add salary to record
  end subroutine update_salary
```

The various types in an inheritance hierarchy are sometimes called subtypes.
Let’s see how this works in a program:

```fortran
program database
  use Student_class
  use Teacher_class
  type (Student) :: studentA  ! declare object
  type (Teacher) :: teacherA  ! declare object

  call new_Student(studentA,1,'Pat','Smith') ! Constructor
  call new_Teacher(teacherA,2,'John','White') ! Constructor

  call add_class(studentA,'Physics 290') ! add class
  call update_salary(teacherA,2000) ! update salary

  call printout(studentA) ! print student record
  call printout(teacherA) ! print teacher record

  call delete(studentA) ! delete student record
  call delete(teacherA) ! delete teacher record
...
```

The output will be

```
1 Pat Smith
Enrolled in: 1 class(es)
Physics 290
2 John White
Salary: 2000
```
Dynamic Dispatch or Run-Time Polymorphism

Fortran95 is very strict about typing. If we write a procedure which works with Student objects, it will not work with Teacher objects. One has to implement two separate procedures, which may be identical except for the type. In this case, Fortran95 is too strict in its typing. It is sometimes useful to be able to use a single type to represent a family of related types.

The purpose of **dynamic dispatch** is to allow one to write generic or abstract procedures which would work on all classes in an inheritance hierarchy, yet produce results which depend on which object was actually used at run-time.

For example, Henderson and Zorn implemented a database class to manage a linked list of various kinds of personnel records. The Database type contains the Personnel record as well as a pointer to the next Database:

```fortran
  type Database
      type (Personnel), pointer :: file ! Personnel data
      type (Database) :: next ! Pointer to next Database
  end type Database
```

The Database class contains methods to add, remove, locate and print records.
For example, here is an implementation of a procedure to add a Personnel record to the end of a linked list of records:

```fortran
subroutine add_record(this,f)
  type (Database), target :: this
  type (Personnel), pointer :: f
  type (Database), pointer :: tmp

  ! traverse database until last associated pointer is found
  tmp => this
  do while (associated(tmp%next))
    tmp => tmp%next
  enddo

  ! store record in last location in linked list
  tmp%file => f

  ! initialize new last location in linked list
  allocate(tmp%next)
  nullify(tmp%next%next)
end subroutine add_record
```

Although this procedure is written for Personnel objects, we would like to be able to use this same procedure for Student and Teacher objects, without having to rewrite it. Object-oriented languages allow this if the types are related by inheritance. Fortran95 does not allow it directly, but it is possible indirectly by writing a special polymorphic class. Dynamic dispatch is the most difficult concept in object-oriented programming to implement in Fortran95.
To implement dynamic dispatch in an OO manner, two features must be constructed: first, a **polymorphic pointer** object which can point to any member of an inheritance hierarchy, and second, a **dispatch mechanism** (or method lookup) which can select the appropriate procedure to execute based on the actual class referenced in the pointer object.

Our polymorphic class contains the following type:

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

which contains pointers to the possible types we wish to support. At any given time, one of these pointers will be associated with an actual object, the others will be nullified. Thus if we create a polymorphic object:

```plaintext
  type (poly_Personnel) :: person
```

then we set either

```plaintext
  person%ps => studentA; nullify(person%pt)
```

or

```plaintext
  person%pt => teacherA; nullify(person%ps)
```

but we do not set both pointers simultaneously.
Selecting which procedure to execute is then decided simply by testing which pointer is associated, as follows:

```fortran
if (associated(person%ps)) then    ! must be a Student
    call printout(person%ps)
else if (associated(person%pt)) then ! must be a Teacher
    call printout(person%pt)
endif
```

One creates a poly_Personnel class to encapsulate each of these two ideas. In this class we define assignment operators to encapsulate the polymorphic type assignment, such as:

```fortran
function assign_student(ps) result(pps)
  type (poly_Personnel) :: pps
  type (Student), target :: ps
  pps%ps => ps                        ! point to Student part
  nullify(pps%pt)                       ! nullify Teacher part
end function assign_student
```

and polymorphic procedures to capsule dynamic dispatch, such as:

```fortran
subroutine print_poly_Personnel(this)
  type (poly_Personnel) :: this
  if (associated(this%ps)) then
    call printout(this%ps)              ! print Student record
  else if (associated(this%pt)) then
    call printout(this%pt)                ! print Teacher record
  endif
end subroutine print_poly_Personnel
```
Once we create such a polymorphic class, it is easy to modify the Database to use the polymorphic type everywhere, enabling it to work with any of the Personnel types.

```fortran
! Personnel data
type Database
  type (poly_Personnel) :: file ! Personnel data
  type (Database) :: next ! Pointer to next Database
end type Database

subroutine add_record(this,f) ! add record
  type (Database), target :: this
  type (poly_Personnel) :: f
  ...

function locate(this,s) result(pps) ! find record
  type (Database), target :: this
  integer :: s
  type (poly_Personnel), pointer :: f
  type (Database), pointer :: tmp
  ! traverse database until ssn is found
  tmp => this
  do while (associated(tmp%next))
    if (getssn(tmp%file==s) then
      pps = tmp%file; return
    endif
    tmp => tmp%next
  enddo
end function locate

The add_record and locate procedures now work for all the types in the inheritance hierarchy.
```
Here is a test program which uses this polymorphic type

```fortran
program database
use Database_class
  type (Database), target :: cs
  type (poly_Personnel) :: person
  type (Student) :: pstudent
  type (Teacher) :: pteacher

  call new_Database(cs)  ! initialize new database

  call new_Student(pstudent,1,'Paul','Jones')
  person = poly(pstudent)  ! make student polymorphic
  call add_record(cs,person)  ! add student to database

  call new_Teacher(pteacher,2,'John','White')
  person = poly(pteacher)  ! make teacher polymorphic
  call add_record(cs,person)  ! add teacher to database

  person = locate(cs,1)  ! find record with ssn = 1
  call printout(person)  ! no matter what its type is
  ...
```
Here is what the polymorphic class looks like:

```fortran
module poly_Personnel
use Student_class
use Teacher_class
type poly_Personnel
    type (Student), pointer :: ps
    type (Teacher), pointer :: pt
end type poly_Personnel
interface poly
    module procedure assign_student, assign_teacher
end interface
interface printout
    module procedure print_poly_Personnel
end interface
contains

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

function assign_teacher(pt) result(pps)
type (poly_Personnel) :: pps
type (Teacher), target :: pt
pps%pt => pt
nullify(pps%ps)
end function assign_teacher
```
To implement such a polymorphic class, one does not need to know anything about what is inside the actual types or the actual methods. Only the interface needs to be known.
In an object-oriented language, such polymorphic classes do not have to be implemented. The parent (base) class behaves as a polymorphic class automatically. The only requirement is that the methods of the subclasses all must have the same interface.

As one can see, faithfully implementing dynamic dispatch in Fortran95 requires a substantial amount of work in writing a special polymorphic class. Although it is straightforward, it can also be tedious. This method of dynamic dispatch emulation is particular useful, if the different subclasses are written by different authors and information hiding is important.

However, for a simple inheritance hierarchy such as this, there is an easier way to emulate dynamic dispatch in Fortran95 if one is willing to sacrifice some amount of information hiding.

Notice that OO languages use the type of the object to make run time decisions about what to execute, and the languages are designed to make this easy to do. It is not so easy in Fortran95, however, because the typing is so strict. Therefore instead of using different types to make decisions, we can use a single type that contains a special flag to indicate what subtype it is and make decisions based on that flag.
To implement this simpler version of dynamic dispatch we create a Personnel type which contains a concatenation of all the components in all the subclasses, as follows:

```fortran
    type Personnel
        integer :: subtype, ssn
        character, dimension(:), pointer :: firstname, lastname
        integer :: nclasses
        type (String), dimension (10) :: classes
        integer :: salary
    end type Personnel
```

This type definition could be in its own module, or in the Personnel module. Notice we are violating information hiding here, because each of the subclasses actually knows about information it should not know about.

All the derived classes now use the Personnel type instead of Student or Teacher. But otherwise, they do not change, except for the constructors, which now include an extra line:

```fortran
    subroutine new_Student(this,s,fn,ln)
        ! Student class constructor
        type (Personnel), intent (out) :: this
        integer, intent (in) :: s
        character*, intent (in) :: fn, ln
        call new_Personnel(this,s,fn,ln)
        this%subtype = 1      ! this represents a Student type
        this%nclasses = 0
    end subroutine new_Student
```
The polymorphic class also no longer has a special type, and its implementation is now much simpler. In fact, it only has two functions in it:

```fortran
 subroutine print_poly_Personnel(this,printssn)
 ! Print Personnel types
 type (Personnel), intent (in) :: this
 logical, optional, intent (in) :: printssn
 if (this%subtype==1) then
   call print_Student(this,printssn)
 else if (this%subtype==2) then
   call print_Teacher(this,printssn)
 endif
 end subroutine print_poly_Personnel
```

and a similar function to delete objects. Procedures such getssn do not have to be reimplemented for each type, since there is now only one type. And similarly, the assignment operators for the polymorphic type are gone. The Database class does not change, except that it uses Personnel types instead of poly_Personnel. The database application is also slightly simpler, since conversions to polymorphic type no longer occur.

This is, in fact, the dynamic dispatch emulation I actually use in my physics applications. However, I discussed the first one, because it is more faithful to the ideas in OOP, which use subtypes, and you should know what these ideas are.