Derived Data Types Outline

1. Derived Data Types Outline
2. Arrays
3. A Company and Its Employees
4. Multiple Employees
5. A New Data Type
6. Breaking Down a Derived Type Definition
7. Declaring an Instance of a Derived Type
8. Extracting Components of an Instance of a Derived Type
9. Extracting Components of an Instance of a Derived Type (continued)
10. Derived Type Example Program
11. Derived Type Example Run
12. An Array of Instances of a Derived Type
13. Array of Derived Type Example Program
14. Array of Derived Type Example Run

See Programming in Fortran 90/95, 1st or 2nd ed, Chapter 8.
Arrays

An array is a collection of values, all of which have the same data type, and all of which have the same essential meaning:

\[ \text{INTEGER, DIMENSION(11) :: OU.score} \]

In memory, the elements of the array are *contiguous*: they occur one after the other in memory.

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
<th>Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345</td>
<td>55</td>
<td>OU_score(1)</td>
</tr>
<tr>
<td>12349</td>
<td>45</td>
<td>OU_score(2)</td>
</tr>
<tr>
<td>12353</td>
<td>42</td>
<td>OU_score(3)</td>
</tr>
<tr>
<td>12357</td>
<td>34</td>
<td>OU_score(4)</td>
</tr>
<tr>
<td>12361</td>
<td>63</td>
<td>OU_score(5)</td>
</tr>
<tr>
<td>12365</td>
<td>41</td>
<td>OU_score(6)</td>
</tr>
<tr>
<td>12369</td>
<td>31</td>
<td>OU_score(7)</td>
</tr>
<tr>
<td>12373</td>
<td>56</td>
<td>OU_score(8)</td>
</tr>
<tr>
<td>12377</td>
<td>35</td>
<td>OU_score(9)</td>
</tr>
<tr>
<td>12381</td>
<td>27</td>
<td>OU_score(10)</td>
</tr>
<tr>
<td>12385</td>
<td>12</td>
<td>OU_score(11)</td>
</tr>
</tbody>
</table>

What if, instead of having a collection of data that all have the same data type and meaning, we had a collection of data that had different data types and meanings?
A Company and Its Employees

Suppose that we work for the Depressingly Dull Corporation (DDC), and our boss tells us to write a program that tracks DDC’s employees. What data will we need?

Well, we’ll probably need to know things like:

- first name;
- last name;
- social security number;
- pay rate;
- number of hours worked this week.

How could we implement this in Fortran 90?

Well, we could simply set up a scalar variable to represent each of these values (and strings for the names):

```fortran
CHARACTER (LEN = maximum_name_length) :: &
  & first_name, last_name
INTEGER :: social_security_number
REAL :: pay_rate
REAL :: hours_worked_this_week
```

Of course, this arrangement would work if our company had exactly one employee. But what if our company has multiple employees?
Multiple Employees

Okay, so suppose that DDC has multiple employees. How could we store the data for them?

Well, we could have an array of each of the pieces of data:

```
CHARACTER (LEN = maximum_name_length), & &
   DIMENSION(number_of_employees) :: & &
   first_name, last_name
INTEGER,DIMENSION(number_of_employees) :: & &
   social_security_number
REAL,DIMENSION(number_of_employees) :: & &
   pay_rate
REAL,DIMENSION(number_of_employees) :: & &
   hours_worked_this_week
```

This approach will work fine, but it’ll be unwieldy to work with. Why? Because it doesn’t match the way that we think about our employees.

That is, we don’t think of having several first names, several last names, several social security numbers and so on.

Instead, we think of having several employees, each of whom has a first name, a last name, a social security number, etc.

In general, it’s much easier to write a program if we can write it in a way that matches the way we think as much as possible.

So: what if we could create a new data type, named Employee, that represented an employee?
A New Data Type

TYPE Employee
   CHARACTER (LEN = maximum_name_length) :: &
     & first_name, last_name
   INTEGER :: social_security_number
   REAL :: pay_rate
   REAL :: hours_worked_this_week
END TYPE Employee

The above declaration creates a new data type, named Employee, consisting of:

- a character string, first_name;
- a character string, last_name;
- an INTEGER scalar, social_security_number;
- a REAL scalar, pay_rate;
- a REAL scalar, hours_worked_this_week.

In Fortran 90, this construct is called a derived type definition, and (not surprisingly) it defines a derived type.

Other languages use different terms: in C the same construct is called a struct definition, in Pascal it’s a record definition, and so on.

The general term for this is a user-defined data type.

NOTE: a derived type definition, as above, only defines the new data type; it does NOT declare any actual data of that type.
Breaking Down a Derived Type Definition

TYPE Employee 
   CHARACTER (LEN = maximum_name_length) :: &
   &      first_name, last_name
   INTEGER :: social_security_number 
   REAL :: pay_rate 
   REAL :: hours_worked_this_week 
END TYPE Employee 

A derived type definition consists of three parts:

- a TYPE statement, which consists of the keyword TYPE followed by the symbolic name that names the derived type;
- a sequence of component definitions, which tell us the pieces of data that constitute an instance of the derived type;
- an END TYPE statement, which is identical to the TYPE statement except that the keyword TYPE is preceded by the keyword END.

(There are other things that can go inside a derived type definition, but we’re not going to deal with them here.)

Note that other languages use different terminology: in C and Pascal, for example, a component is called a field of the user-defined data type.
Declaring an Instance of a Derived Type

```fortran
TYPE Employee
    CHARACTER (LEN = maximum_name_length) :: &
        first_name, last_name
    INTEGER :: social_security_number
    REAL :: pay_rate
    REAL :: hours_worked_this_week
END TYPE Employee
```

The above derived type definition defines the derived type named `Employee`, but does not declare any data of the `Employee` type.

To declare an instance of an `Employee`, we need a derived type declaration:

```fortran
TYPE(Employee) :: worker_bee
```

The above statement declares that `worker_bee` is an instance of type `Employee`; it tells the compiler to grab a group of bytes, name them `worker_bee`, and think of them as storing an `Employee`.

How many bytes?

That depends on the platform and the compiler, but assuming that a `CHARACTER` takes 1 byte, an `INTEGER` takes 4 bytes and a `REAL` takes 4 bytes, and assuming a named constant declaration

```fortran
INTEGER,PARAMETER :: maximum_name_length = 32
```

then the above declaration of an instance of `Employee` would take up 76 bytes:

\[
1 \cdot 32 + 1 \cdot 32 + 4 + 4 + 4 = 76
\]
Extracting Components of an Instance of a Derived Type

INTEGER, PARAMETER :: maximum_name_length = 32

TYPE Employee
  CHARACTER (LEN = maximum_name_length) :: &
  & first_name, last_name
  INTEGER :: social_security_number
  REAL :: pay_rate
  REAL :: hours_worked_this_week
END TYPE Employee

TYPE(Employee) :: worker_bee

Okay, so now we have an instance of type Employee named worker_bee.

But how can we use the values of its component data? For example, how do we get the social security number of worker_bee?

To use an individual component of a derived type, we use the component operator, which is the percent sign (%):
worker_bee%social_security_number

For example, we can assign a value to the social security number of worker_bee:
worker_bee%social_security_number = 123456789

Likewise, we can output the social security number of worker_bee:
PRINT *, worker_bee%social_security_number
Extracting Components of an Instance of a Derived Type (continued)

INTEGER, PARAMETER :: maximum_name_length = 32
TYPE Employee
  CHARACTER (LEN = maximum_name_length) :: &
    first_name, last_name
  INTEGER :: social_security_number
  REAL :: pay_rate
  REAL :: hours_worked_this_week
END TYPE Employee

TYPE(Employee) :: worker_bee

We said that we can use the component operator (%) to get an individual component of an instance of a derived type:

worker_bee%social_security_number = 123456789
PRINT *, worker_bee%social_security_number

Notice that this usage is analogous to the use of array indices:

OU_score(1) = 55
PRINT *, OU_score(1)

In the case of arrays, we said that an individual element of an array behaves exactly like a scalar of the same basic type.

Likewise, a component of a derived type behaves exactly like a variable of the same type as the component.

Thus, worker_bee%social_security_number can be used exactly like an INTEGER scalar, and worker_bee%first_name can be used exactly like a character string of length 32.
Program employee_test

IMPLICIT NONE
INTEGER, PARAMETER :: maximum_name_length = 32

TYPE Employee
  CHARACTER (LEN = maximum_name_length) :: first_name, last_name
  INTEGER :: social_security_number
  REAL :: pay_rate
  REAL :: hours_worked_this_week
END TYPE Employee

TYPE(Employee) :: worker_bee
REAL :: worker_bee_pay

PRINT *, "What is the first name of the employee?" 
READ *, worker_bee%first_name
PRINT *, "What is the last name of the employee?" 
READ *, worker_bee%last_name
PRINT *, "What is the social security number ", &
& "of the employee?"
READ *, worker_bee%social_security_number
PRINT *, "What is the pay rate of the employee?" 
READ *, worker_bee%pay_rate
PRINT *, "How many hours did the employee work ", &
& "this week?"
READ *, worker_bee%hours_worked_this_week

worker_bee_pay = &
& worker_bee%pay_rate * &
& worker_bee%hours_worked_this_week

PRINT "(A,A,A,A,A,I9,A)" , &
& "Employee ", &
& TRIM(worker_bee%first_name), " ", &
& TRIM(worker_bee%last_name), " (", &
& worker_bee%social_security_number, ")"

PRINT "(A,F5.2,A)" , " worked ", &
& worker_bee%hours_worked_this_week, &
& " hours this week"
PRINT "(A,F5.2,A)" , " at a rate of ", &
& worker_bee%pay_rate, " per hour," 
PRINT "(A,F6.2,A)" , " earning ", &
& worker_bee_pay, "."
Derived Type Example Run

% f95 -o employee_test employee_test.f90
% employee_test
What is the first name of the employee?
Henry
What is the last name of the employee?
Neeman
What is the social security number of the employee?
123456789
What is the pay rate of the employee?
6.75
How many hours did the employee work this week?
39.5
Employee Henry Neeman (123456789) worked 39.50 hours this week at a rate of $ 6.75 per hour, earning $266.62.
An Array of Instances of a Derived Type

When we started working on this task, we wanted to work out a convenient way to store the many employees of the Depressingly Dull Corporation (DDC).

So far, we’ve worked out how to defined a derived data type, how to declare an individual instance of the derived type, and how to use the components of the instance.

So, how would we declare and use an array of instances of a derived type?

```fortran
TYPE(Employee),DIMENSION(maximum_employees) :: &
& worker_bee_array
```

Not surprisingly, an array whose elements are a derived type can also be declared ALLOCATABLE:

```fortran
TYPE(Employee),DIMENSION(:),ALLOCATABLE :: &
& worker_bee_array
```

An individual element of such an array can be accessed using indexing, exactly as if it were an element of an array of scalar type:

```fortran
worker_bee_array(index)
```

Components of an individual element of an array of a derived type can be accessed thus:

```fortran
worker_bee_array(index)%pay_rate
```

For example:

```fortran
worker_bee_array(index)%pay_rate = 6.50
PRINT *, worker_bee_array(index)%pay_rate
```
Array of Derived Type Example Program

PROGRAM employee_array_test
IMPLICIT NONE
INTEGER,PARAMETER :: maximum_name_length = 32
TYPE Employee
  CHARACTER (LEN = maximum_name_length) :: &
    first_name, last_name
  INTEGER :: social_security_number
  REAL :: pay_rate
  REAL :: hours_worked_this_week
END TYPE Employee
TYPE(Employee),DIMENSION(:),ALLOCATABLE :: &
  worker_bee
REAL,DIMENSION(:),ALLOCATABLE :: worker_bee_pay
INTEGER :: number_of_worker_bees, index
PRINT *, "How many employees does the company have?"
READ *, number_of_worker_bees
ALLOCATE(worker_bee(number_of_worker_bees))
ALLOCATE(worker_bee_pay(number_of_worker_bees))
DO index = 1, number_of_worker_bees
  PRINT *, "What is the first name of employee #", index, "?"
  READ *, worker_bee(index)%first_name
  PRINT *, "What is the last name of the employee #", index, "?"
  READ *, worker_bee(index)%last_name
  PRINT *, "What is the social security number ", &
    "of the employee ", index, "?"
  READ *, worker_bee(index)%social_security_number
  PRINT *, "What is the pay rate of the employee ", index, "?"
  READ *, worker_bee(index)%pay_rate
  PRINT *, "How many hours did the employee ", &
    "work ", "this week?"
  READ *, worker_bee(index)%hours_worked_this_week
END DO !! index = 1, number_of_worker_bees
DO index = 1, number_of_worker_bees
  worker_bee_pay(index) = worker_bee(index)%pay_rate * &
    worker_bee(index)%hours_worked_this_week
END DO !! index = 1, number_of_worker_bees
DO index = 1, number_of_worker_bees
    TRIM(worker_bee(index)%first_name), " ", &
    TRIM(worker_bee(index)%last_name), " ", &
    worker_bee(index)%social_security_number, ")"
  PRINT "(A,F5.2,A)", " worked ", &
    " hours this week"
  PRINT "(A,F5.2,A)", " at a rate of ", &
    worker_bee(index)%pay_rate, " per hour,"
  PRINT "(A,F6.2,A)", " earning ", &
    worker_bee_pay(index), "."
END DO !! index = 1, number_of_worker_bees
END PROGRAM employee_array_test
Array of Derived Type Example Run

% f95 -o employee_array_test employee_array_test.f90
% employee_array_test

How many employees does the company have?
2
What is the first name of employee # 1?
Henry
What is the last name of the employee # 1?
Neeman
What is the social security number of the employee # 1?
123456789
What is the pay rate of the employee # 1?
6.75
How many hours did the employee # 1 work this week?
39.5
What is the first name of employee # 2?
Lee
What is the last name of the employee # 2?
Kim
What is the social security number of the employee # 2?
987654321
What is the pay rate of the employee # 2?
8.50
How many hours did the employee # 2 work this week?
22.75
Employee Henry Neeman (123456789)
worked 39.50 hours this week
at a rate of $ 6.75 per hour,
earning $266.62.
Employee Lee Kim (987654321)
worked 22.75 hours this week
at a rate of $ 8.50 per hour,
earning $193.38.