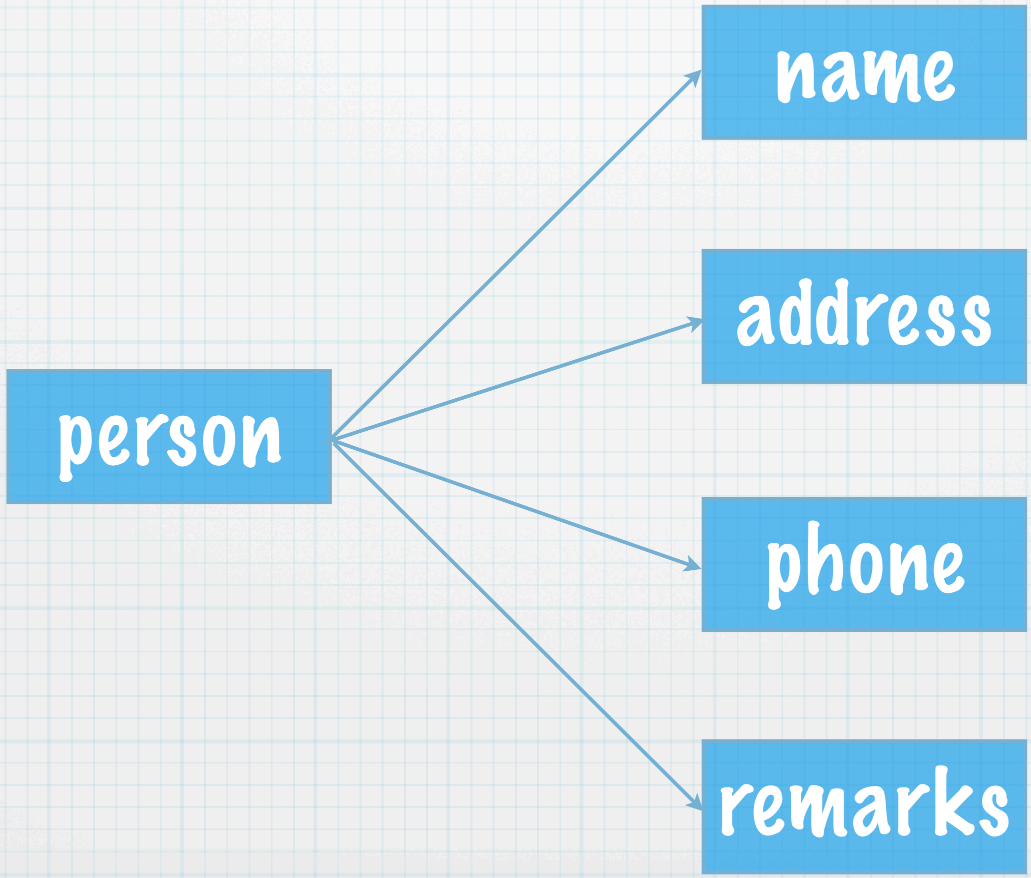
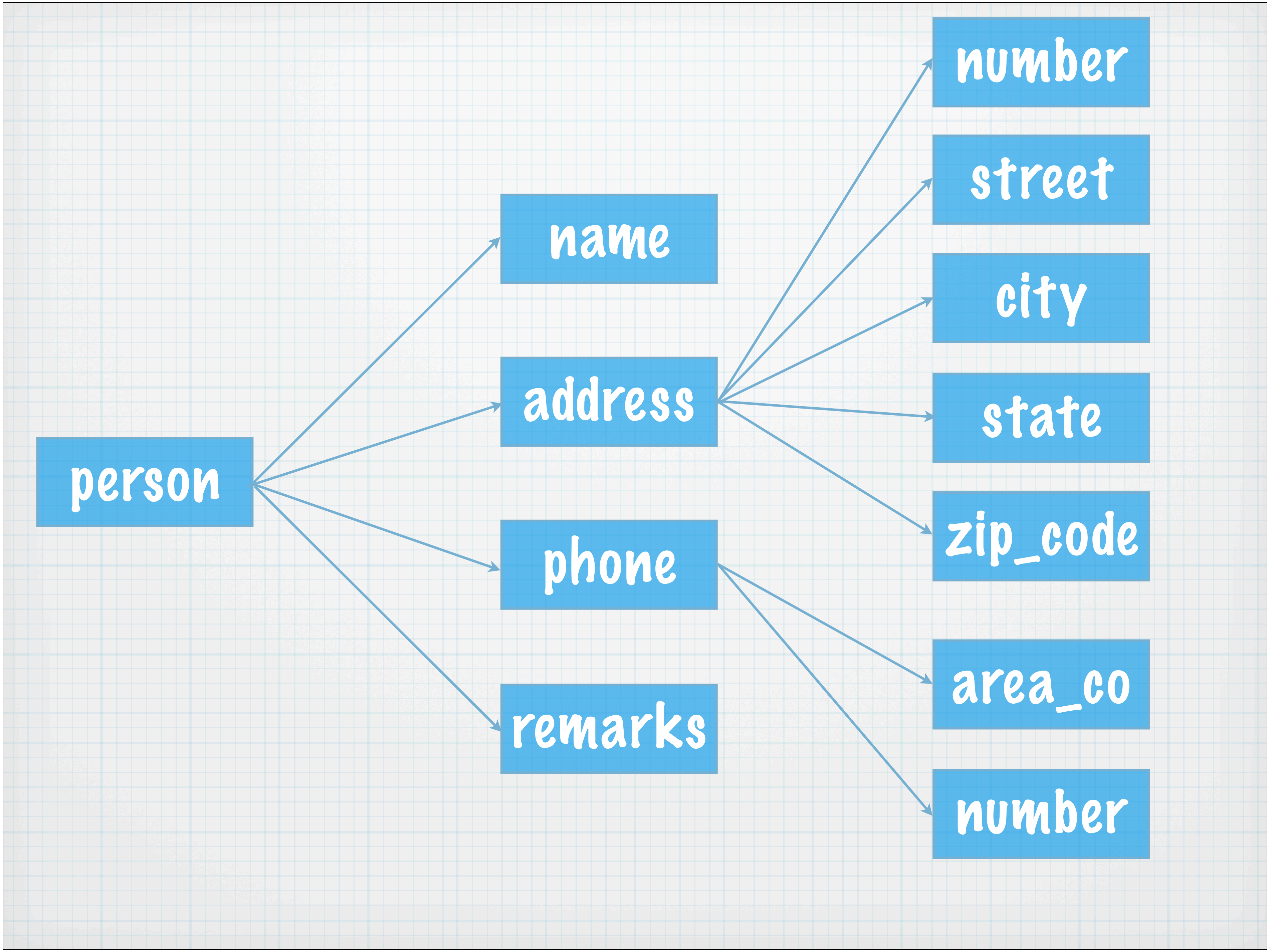
Structures and Derived Types

Introduction

It's often useful to group related variables or components into a single entity or **structure**, and these may even be comprised of objects of different types.





person
name
address
phone
remarks
number
street
city
state
zip_code
area_co
number

First we define the various types of our structure:

```
type phone_type
   integer :: area_code, number
end type phone_type

type address_type
   integer :: number
   character (len = 30) :: street, city
   character (len = 2) :: state
   integer :: zip_code
end type address_type

type person_type
   character (len = 40) :: name
   type (address_type) :: address
   type (phone_type) :: phone
   character (len = 100) :: remarks
end type person_type
```

Since phone_type and address_type were defined before person_type, we could use them as components of the person_type structure.

Declaring and Using Structures

Now we can define a variable using our new derived type:

```
type (person_type) :: joan
type (person_type), dimension(1000) :: black_book
```

Note the difference between a **type definition** and a **type declaration**.

Also, the component names are local to the structure, so there is no problem if the same program unit also uses simple variables like number, street, city, etc.

* The only thing you can't put into a derived type is an allocatable array, but you can use a pointer to achieve exactly the same thing.

Referencing Structure Components

Write the name of the structure followed by a % and then the name of the component:

```
joan % address            ! blanks are permitted but not required
joan % address % state
joan % phone % area_code
black_book(42) = joan     ! copy all components
black_book(42) % address % number = joan % address % number + 1
```

Note the difference between a **type definition** and a **type declaration**.

Let's look at an example of how structures could be used in a program. Suppose we want to print out the names of all persons who live in a given zip code:

```
subroutine find_zip (zip)

   integer, intent(in) :: zip
   integer :: entry

   do entry = 1, number_of_entries
      if (black_book(entry) % address % zip_code == zip) then
         print *, black_book(entry) % name
      endif
   enddo

end subroutine find_zip
```

Structure Constructors

Each derived-type definition creates a constructor whose name is the same as that of the derived type, and it can be used to create a structure of the named type.

```
joan % phone = phone_type(505, 2750800)
```

It is not necessary that the function arguments be constants:

```
joan = person_type("Joan Doe", john % address, &
            phone_type(505, fax_number - 1), &
            "Same address as husband John")
```

A "real world" example from the CSU global couple model (and a teaser):

```
type, public :: qp_type
   integer (kind=int_kind) :: itag
   character (len=30) :: name
   character (len=30) :: units
   character (len=80) :: descr
   integer (kind=int_kind) :: nsamples
   logical (kind=log_kind) :: log
   logical (kind=log_kind) :: amip_sampling
   real (kind=real_kind), pointer :: qp2_data(:,:,:)
   real (kind=real_kind), pointer :: qp3_data(:,:,:,:)
 end type
```

So you can't use an allocatable (dynamic) array within a structure, but you can effectively do it using a pointer array.

Modules and Interfaces

Introduction

* Passing arguments is not always the most effective way to share a large number of variables among many different procedures, and on some systems may actually reduce efficiency.
* Modules provide another way of sharing constants, variables and type definitions.
* They also provide a way of sharing procedures, which is useful when building a library of data and procedures that can be accessible to many different programs.
* A **module** is a program unit that is not executed directly, but contains data specifications and procedures that may be utilized via the **use** statement.

Basic Layout

```
module nameOfModule
   implicit none
!-- declare data and interface statements
   contains
!-- subroutines and functions are declared here
end module nameOfModule

!-- use a module
program mainProgram
   use nameOfModule         ! must be first
   implicit none
   .
   .
end program mainProgram
```

A simple example:

```
module trig_constants
  implicit none
  real, parameter :: pi = 3.1415926, rtod = 180.0/pi, dtor = pi/180.0
end module trig_constants

program calculate
  use trig_constants
  implicit none
  real :: angle = 30.0
  write(*,*) sin(angle*dtor)
end program calculate
```

- USE statements always precede all other types of specification, including IMPLICIT NONE.
- The module must be compiled before all other program units which use it.
- Why not just use an include statement instead?

Advantages of Modules

* Module procedures can be accessed by the main program as well as any other module and procedures.
* We can control accessibility of data and procedures.
  * **use** some_module, only : x, y, z
  * also **public** and **private** statements/attributes
* We can avoid name clashes.
  * **use** some_module, nu => nr_of_unknowns

Combo:

  * **use** some_module, only : dbl => double, quad

* The interface of module procedures is automatically explicit. This means that the compiler can check actual and dummy arguments for consistency. Also, we need explicit interfaces to use "advanced features" like assumed-shape arrays, pointer arrays, optional arguments, user-defined operators, etc.
    * see **badpass.f90**, **goodpass1.f90**, etc.