Composite Types

What For?

- Arrays are fast and very useful for certain types of code, but they are limited.
  - In many languages they must be of fixed size
  - Every element must be the same type.
- What about data that is logically connected but not of the same type?

Simple Composite Types

- Many scripting languages have one or more built-in types that can hold arbitrary elements.
- Python: lists, dictionaries, sets
- Matlab: cell arrays
- R: lists, dataframes
- So we can have a list like

```
L:=[“Spam”, “Spam”, 2, “Eggs”, “Spam”]
```

More Composite Types

- Some languages like Python make a distinction between ordered types (lists, tuples, NumPy arrays) and unordered types (dictionaries, sets).
- Elements of ordered types can be accessed by an index just as for arrays.

```
print L[2]
```

```
2
```

(zero based indexing)

Programmer-Defined Composite Types

- Nearly all (recent) languages provide a way for a programmer to define his/her own composite type.
- This allows related data to be *encapsulated* into the type.
- Defined types are the foundation of object-oriented programming.
- Defined types make construction of more advanced data structures (linked lists, trees) in compiled languages much easier.

Example

- Every textbook's favorite example: payroll
- For each employee you have data such as
  - Name (string)
  - Department (string)
  - Salary (number)
  - ID (number)
  - and so forth.

Payroll (cont.)

- We could write a payroll system with a clunky arrangement of standard arrays, and with “tables” to match indices of one array to indices of another.
- E.g.
- names(nemp)
- dept(nemp)
- salary(nemp)

- Then process with

```
for i = 1 to nemp do
   if ( name[i] == "Eric" )
      salary[i]:=salary[i]*raise[i]
   end if
end for
```

There is nothing here to associate salary directly with employee name. We must make sure their positions in the different arrays match.

A Better Approach

- Most languages have something called by a variety of names like defined type, structure, etc. Computer scientists often call this a *record type*. The individual types that make it up are usually called *fields* or *members*.

```
record employee
   string name
   string department
   float salary
end record
```

Record Types

- We may now declare variables of type “employee”

```
employee :: manager, custodian
```

- We access the fields with a separator (most languages use a period, Fortran uses a percent sign).

```
manager.name:=“Andrew”
manager.department:=“UVACSE”
manager.salary:=120000.
```

Arrays of Composite Types

- Most languages permit arrays or lists with each element a variable of a given composite type.
- E.g. transfer an employee:

```
employee[nemps] :: employees
for i from 1 to nemps do
   if  (employees[i].name=="Fred")
      employees[i].department:="Accounting"
   end if
end for
```

Records of Records

- Nearly all languages permit members of a record type to be themselves records.

```
record department
     string          :: department_name
     employee(nm)    :: managers
      employee(nw)    :: worker_bees
     integer         :: max_employees
end record
```

- The program unit that defines `department` must have access to a definition of `employee`.

Example

```
department accounting
accounting.managers[1].name:=“Boss”
accounting.managers[1].salary:=200000.
for i from 1 to nw do
   accounting.worker_bee[i].salary:=50000.
  if ( i != 1 )
     accounting.managers[i].salary:=100000.
  end if
end for
```

Variable Scope in Types

- Variables defined as members of a record type are *local* to the type and must be accessed through the *instance* (variable) of the type.
- Thus you may have a type `animal` with members `species, genus, order, class` and you can set up a particular instance of the type as:

```
animal :: function newanimal(species,genus,order,class)
    newanimal.species=species
    newanimal.genus=genus
    newanimal.order=order
    newanimal.class=class
    return newanimal
end function newanimal
```

- Even with the same names, species, genus, etc. do not conflict with the names of the members.

Information Hiding

- Defined types help us to *encapsulate* the parts of the program that are most likely to change into a unit that presents a (reasonably) unchanging interface to the rest of the program.
- For example, if I add another member to the `animal` class, I do not have to change any of the code that does something with the existing members.

Type Polymorphism

- Polymorphism is when a method/procedure/ subprogram can operate on multiple types.
- Depending on the language, we may be able to *overload* procedures and even operators to work with user-defined types.

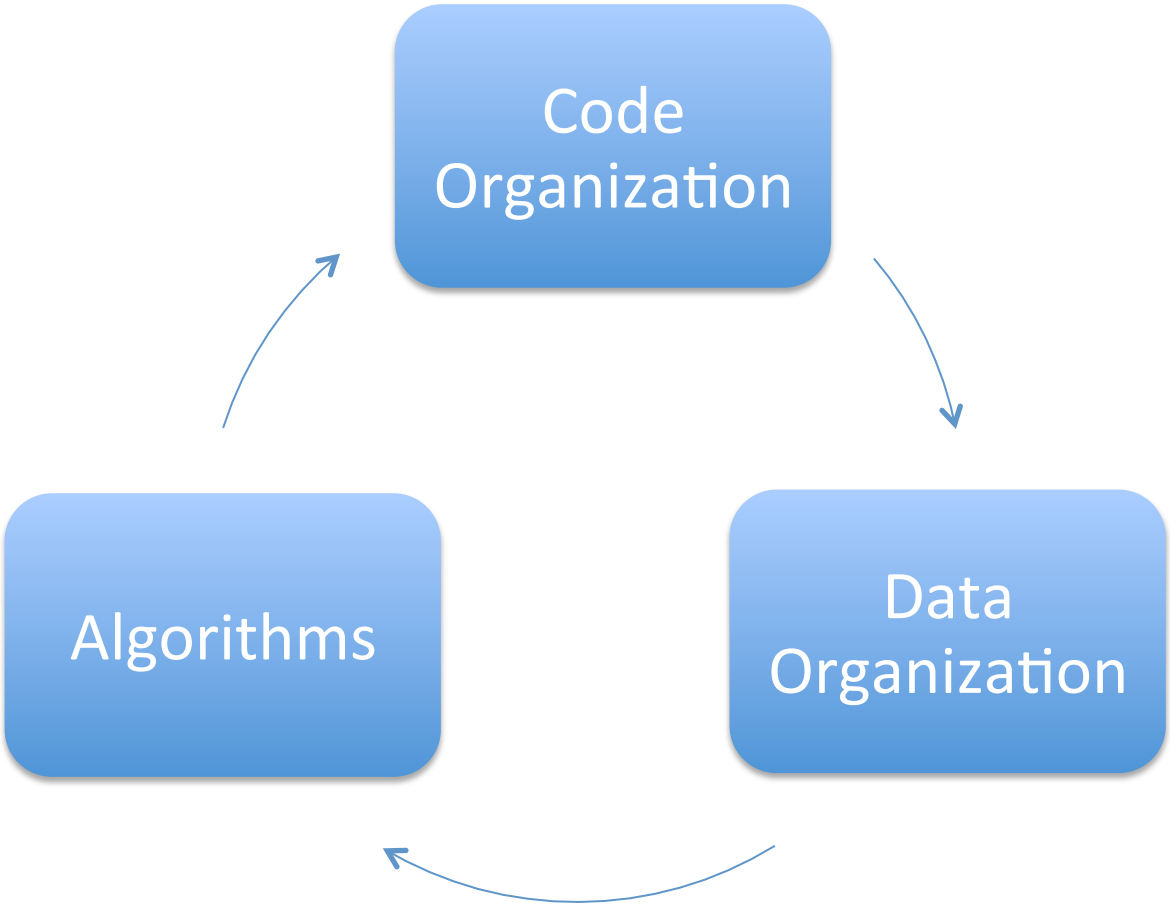
Modular Programming

- In modular programming, programs are broken into modules (explicitly, if available in the language, or it can be imitated by code organization in languages that lack it). A module has a particular purpose and presents an interface to the rest of the program. Thus it can implement information hiding.
- Similar to object-oriented programming – a module has properties similar to that of an object, but it cannot be *instantiated* (declared as a variable) explicitly.
- Modules are frequently built around defined types.

Separation Of Concerns

- Modular (and OO) programming is a way to separate “concerns.”
- Only the module needs to deal with the details of its “concern” – the rest of the code communicates via the interface presented by the module.

Code Design



Data Organization

- Your choice of data types and structures can limit your algorithm choices and vice versa. Think about your structures *before* you start coding, especially for larger projects.
- Code correctness and maintainability are the goals.