Types

- What is a type?
- Type checking
- Type conversion
- Aggregates: strings, arrays, structures
- Enumeration types
- Subtypes

What is a type?

- A set of values and the valid operations on those values
  - E.g., integers + - * *div* < <= = >= > ...
  - Enables programmers to think of modelling reality with different kinds of values
  - Program semantics (meaning) embedded in types used
  - Additional correctness check provided beyond valid syntax

Types

- **Implicit**
  - – **If variables are typed by usage**
    - **Prolog, Scheme, Lisp, Smalltalk**
- **Explicit**
  - – **If declarations bind types to variables at compile time**
    - **Pascal, Algol68, C, C++, Java**
- **Mixture**
  - – **Implicit by default but allows explicit declarations**
    - **Haskell, ML**

Types of Expressions

- **If *f* has type *S* → *T* and *x* has type *S*, then *f(x)* has type *T***
  - – **type of `3 div 2` is *int***
  - – **type of `round(3.5)` is *int***
- ***Type error* - using wrongly typed operands in an operation**
  - – `round(“Nancy”)`
  - – `3.5 div 2`
  - – `“abc”+ 3`

Type Checking

- *Goal:* to find out as early as possible, if each procedure and operator is supplied with the correct type of arguments
- Modern PLs often designed to do type checking (as much as possible) during compilation

Type Checking

- *Compile-time* (static)
  - At compile-time, uses declaration information or can infer types from variable uses
- *Runtime* (dynamic)
  - During execution, checks type of object before doing operations on it
    - Uses type tags to record types of variables

Type Safety

- A *type safe* program executes on all inputs without type errors
  - Goal of type checking is to ensure type safety
  - Type safe does not mean without errors

```
read n;
if n>0 then y:=“ab”;
            if n<0 then x := y-5;
```

- Note that assignment to **x** is never executed so program is *type safe* (but contains an error).

Strong Typing

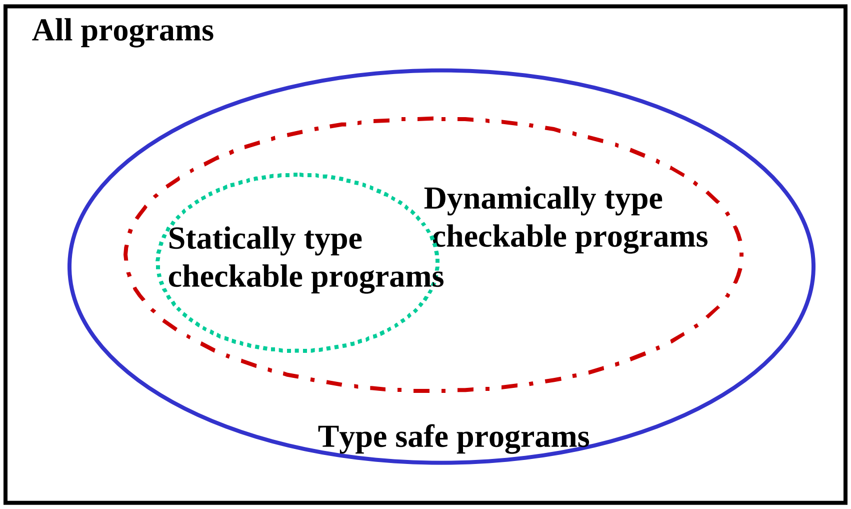
- *Strongly typed PL* By definition, PL requires all programs to be type checkable
- *Statically strongly typed PL* - compiler allows only programs that can be type checked fully at compile-time
  - Algol68, ML
- *Dynamically strongly typed PL* -Operations include code to check runtime types of operands, if type cannot be determined at compile-time
  - Pascal, Java

Type Checking

- Kind of typing used is orthogonal to when complete type checking can be accomplished.

	static checking	dynamic checking
Implicit types	ML	Scheme
Explicit types	Algol68	C, Pascal

Hierarchy of Programs



All programs

Dynamically type checkable programs

Statically type checkable programs

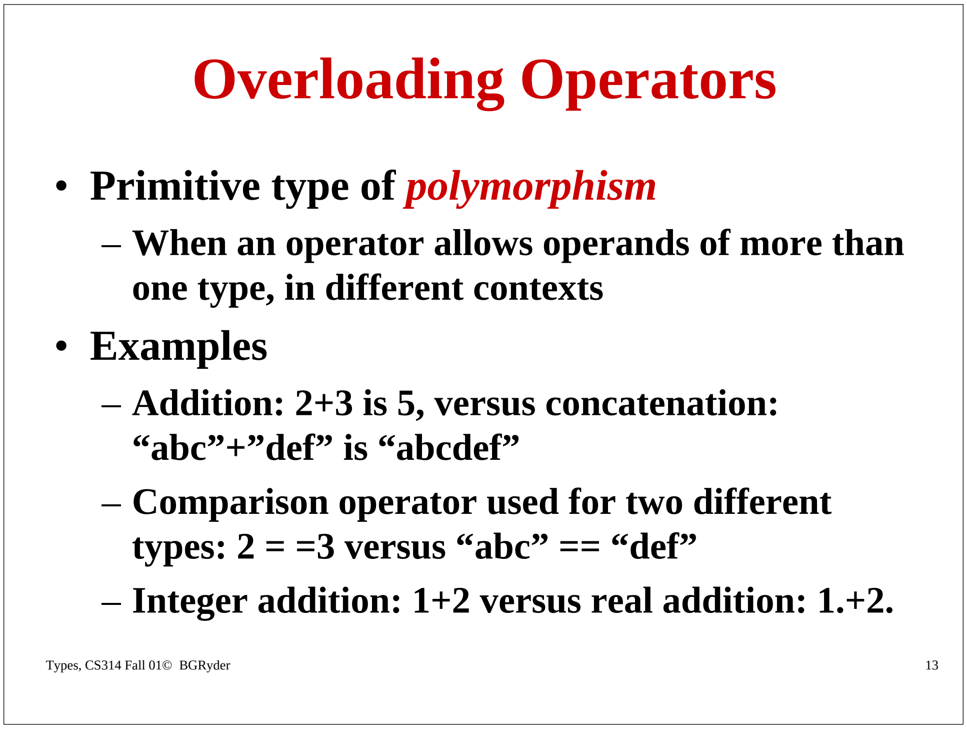
Type safe programs

Type Conversion

- Implicit conversion - *coercion*
  - In C, mixed mode numerical operations
    - `double d,e;…e=d+2;//`2 coerced to 2.0
  - Usually can use *widening* or conversion without loss of precision
    - integer → double, float → double
    - But real → int may lose precision and therefore cannot be implicitly coerced!
  - Cannot coerce user-defined types or structures

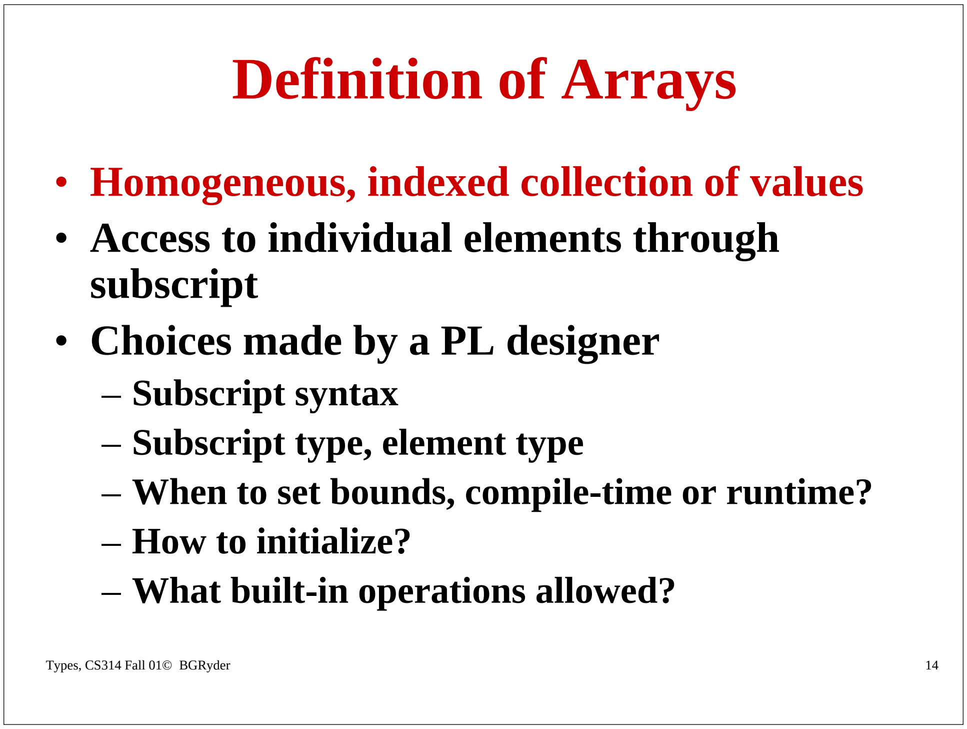
Type Conversion

- Explicit conversion
  - In Pascal, can explicitly convert types which may lose precision (*narrowing*)
    - `round(s)` real → int by rounding
    - `trunc(s)` real → int by truncating
  - In C, casting sometimes is explicit conversion
    - `dqstr((double) n)`where `n` is declared to be an `int`
    - `freelist *s; ... (char *) s;` forces `s` to be considered as pointing to a char for purposes of pointer arithmetic



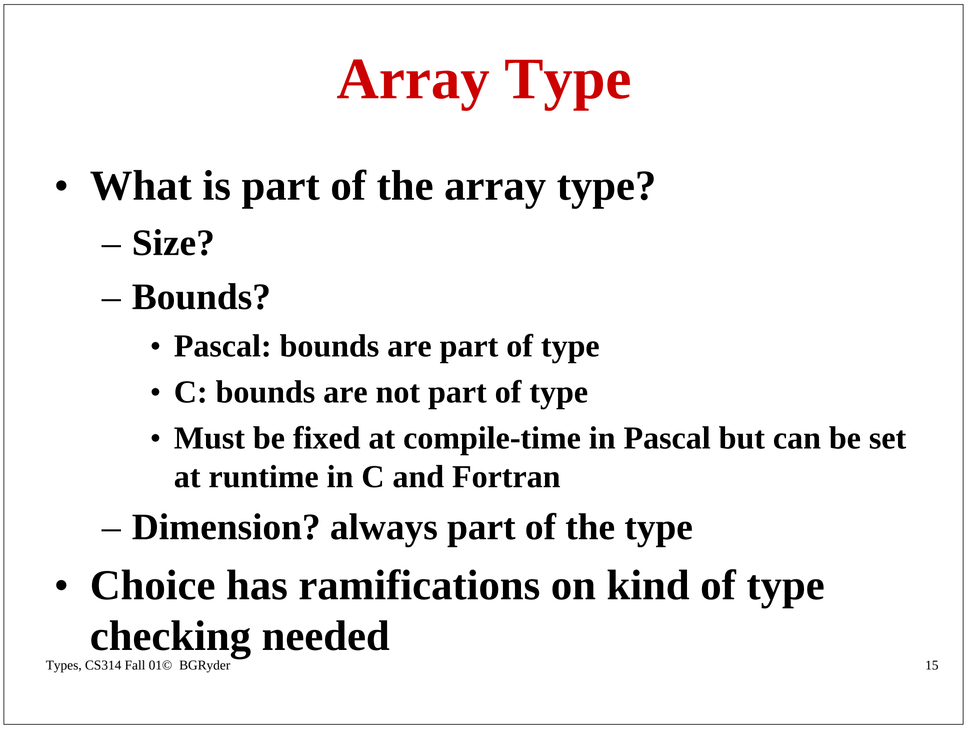
Overloading Operators

- Primitive type of *polymorphism*
  - When an operator allows operands of more than one type, in different contexts
- Examples
  - Addition: 2+3 is 5, versus concatenation: “abc”+”def” is “abcdef”
  - Comparison operator used for two different types: 2 = =3 versus “abc” == “def”
  - Integer addition: 1+2 versus real addition: 1.+2.



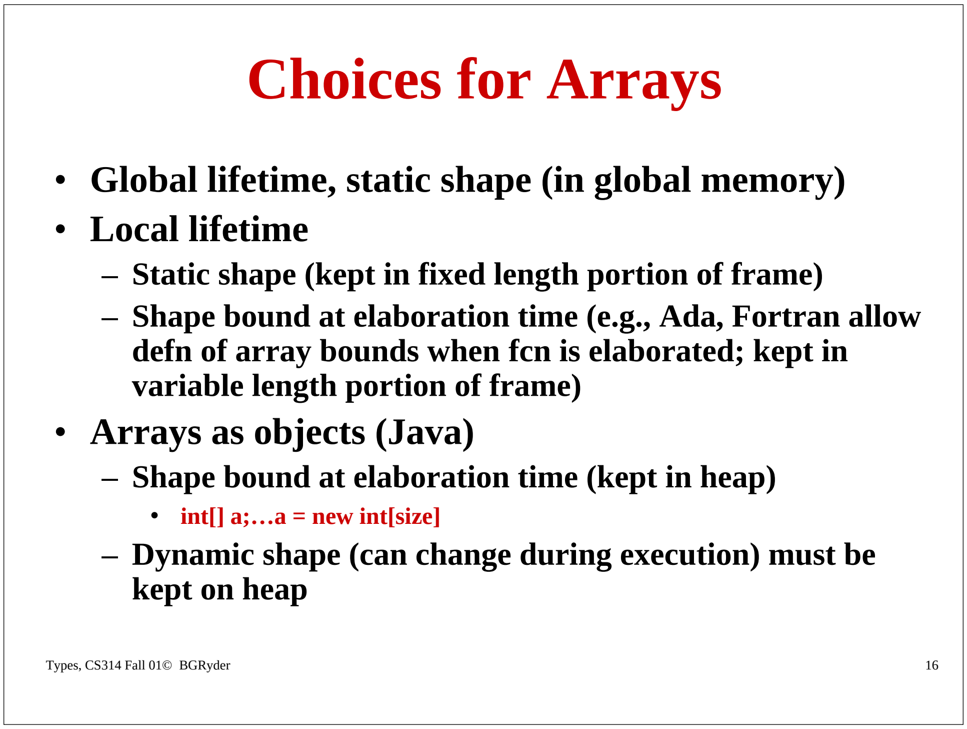
Definition of Arrays

- Homogeneous, indexed collection of values
- Access to individual elements through subscript
- Choices made by a PL designer
  - Subscript syntax
  - Subscript type, element type
  - When to set bounds, compile-time or runtime?
  - How to initialize?
  - What built-in operations allowed?



Array Type

- What is part of the array type?
  - Size?
  - Bounds?
    - Pascal: bounds are part of type
    - C: bounds are not part of type
    - Must be fixed at compile-time in Pascal but can be set at runtime in C and Fortran
  - Dimension? always part of the type
- Choice has ramifications on kind of type checking needed

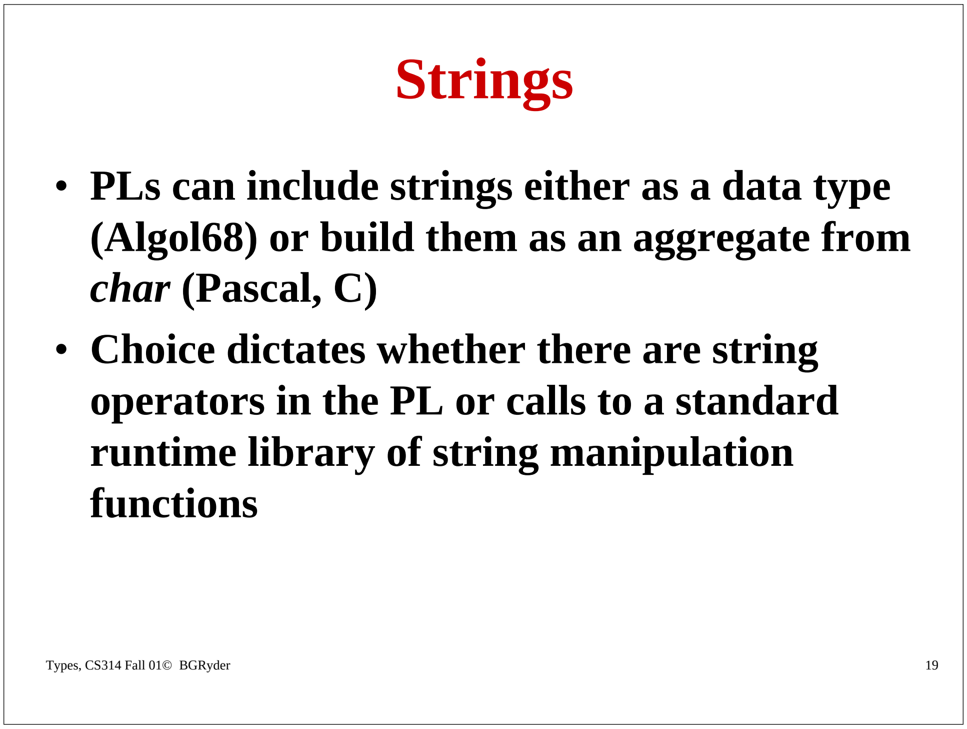


Choices for Arrays

- Global lifetime, static shape (in global memory)
- Local lifetime
  - Static shape (kept in fixed length portion of frame)
  - Shape bound at elaboration time (e.g., Ada, Fortran allow defn of array bounds when fcn is elaborated; kept in variable length portion of frame)
- Arrays as objects (Java)
  - Shape bound at elaboration time (kept in heap)
    - int[] a;…a = new int[size]
  - Dynamic shape (can change during execution) must be kept on heap

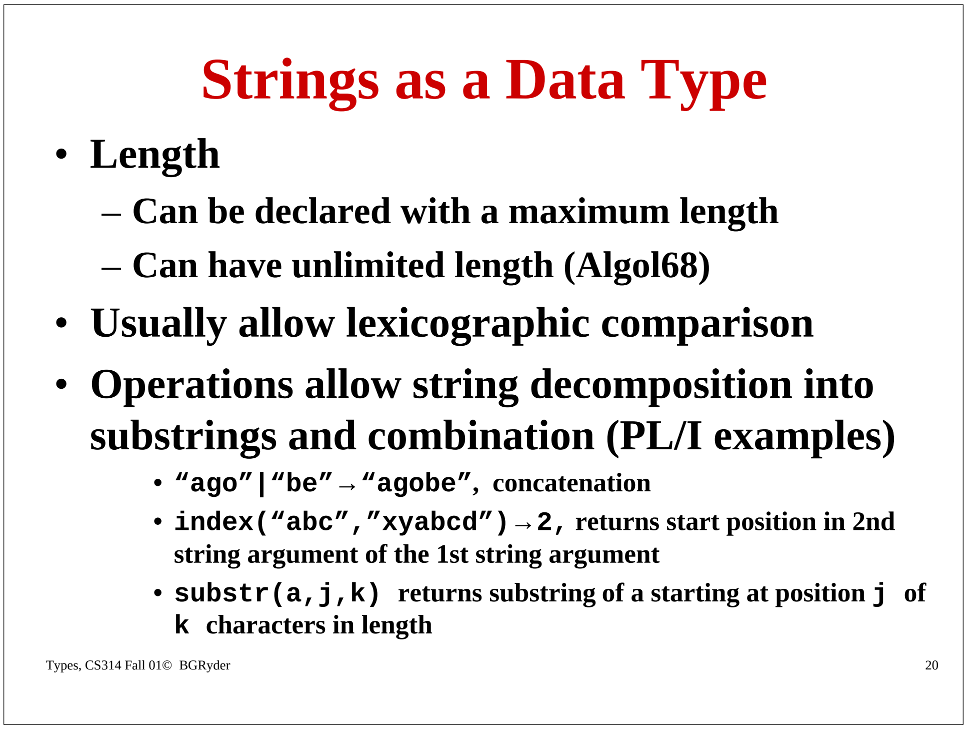
Arrays - Dope Vector

- For arrays whose length is not knowable at compile-time, we use a descriptor of fixed size on the frame, and then allocate space for the array data separately
- Dope vector contains:
  - Name, type of subscript, bounds, type of elements, number of bytes in each element, pointer to first storage location of array
  - Allows calculation of actual frame address of an array element from these values (more next lecture)



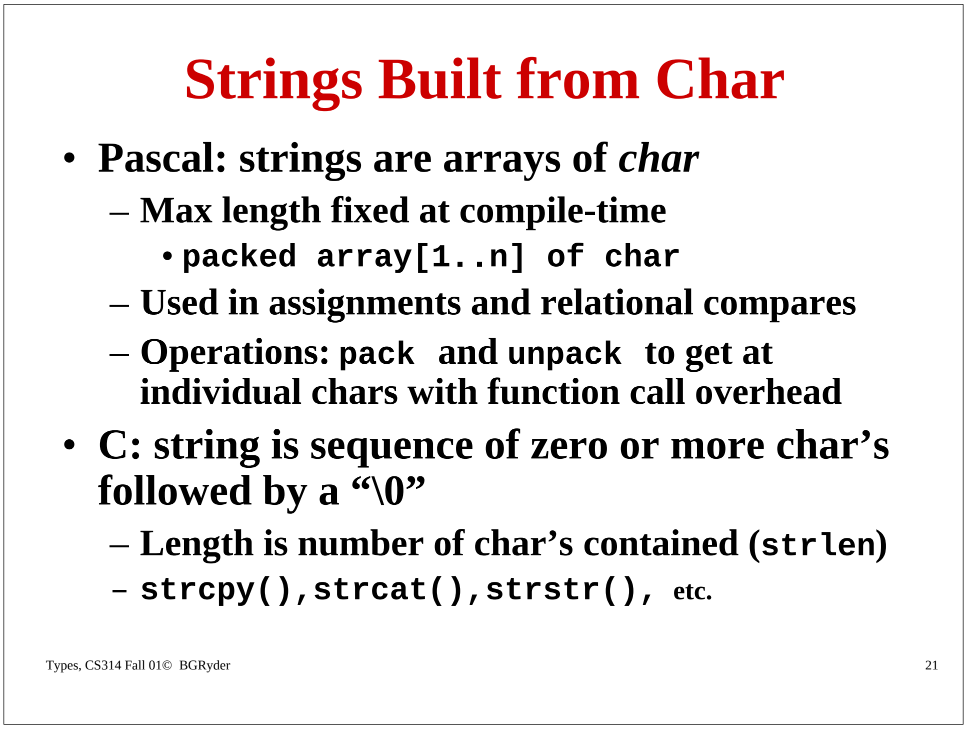
Strings

- PLs can include strings either as a data type (Algol68) or build them as an aggregate from *char* (Pascal, C)
- Choice dictates whether there are string operators in the PL or calls to a standard runtime library of string manipulation functions



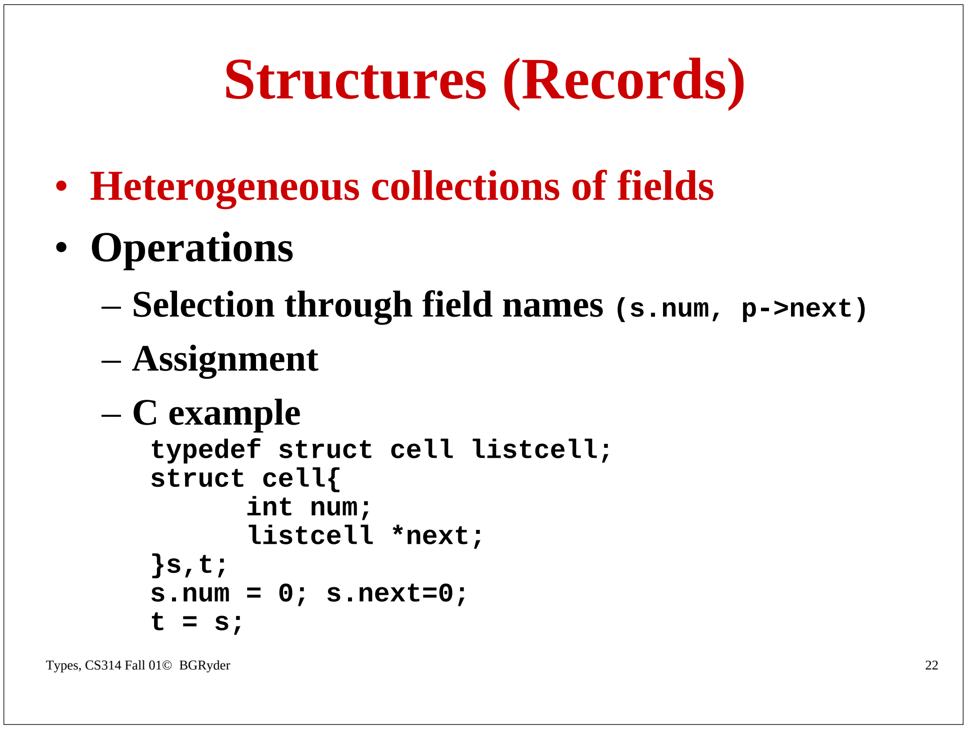
Strings as a Data Type

- Length
  - Can be declared with a maximum length
  - Can have unlimited length (Algol68)
- Usually allow lexicographic comparison
- Operations allow string decomposition into substrings and combination (PL/I examples)
  - `“ago”|“be”`→`“agobe”`, concatenation
  - `index(“abc”,”xyabcd”)`→`2`, returns start position in 2nd string argument of the 1st string argument
  - `substr(a,j,k)` returns substring of a starting at position `j` of `k` characters in length



Strings Built from Char

- **Pascal: strings are arrays of *char***
  - **Max length fixed at compile-time**
    - `packed array[1..n] of char`
  - **Used in assignments and relational compares**
  - **Operations: `pack` and `unpack` to get at individual chars with function call overhead**
- **C: string is sequence of zero or more char's followed by a "\0"**
  - **Length is number of char's contained (`strlen`)**
  - `strcpy(),strcat(),strstr(),` **etc.**



Structures (Records)

- **Heterogeneous collections of fields**
- **Operations**
  - **Selection through field names** `(s.num, p->next)`
  - **Assignment**
  - **C example**

```
typedef struct cell listcell;
struct cell{
      int num;
      listcell *next;
}s,t;
s.num = 0; s.next=0;
t = s;
```

Enumeration Type

- Ordered sequence of literal values
- Operations - assignment, comparison

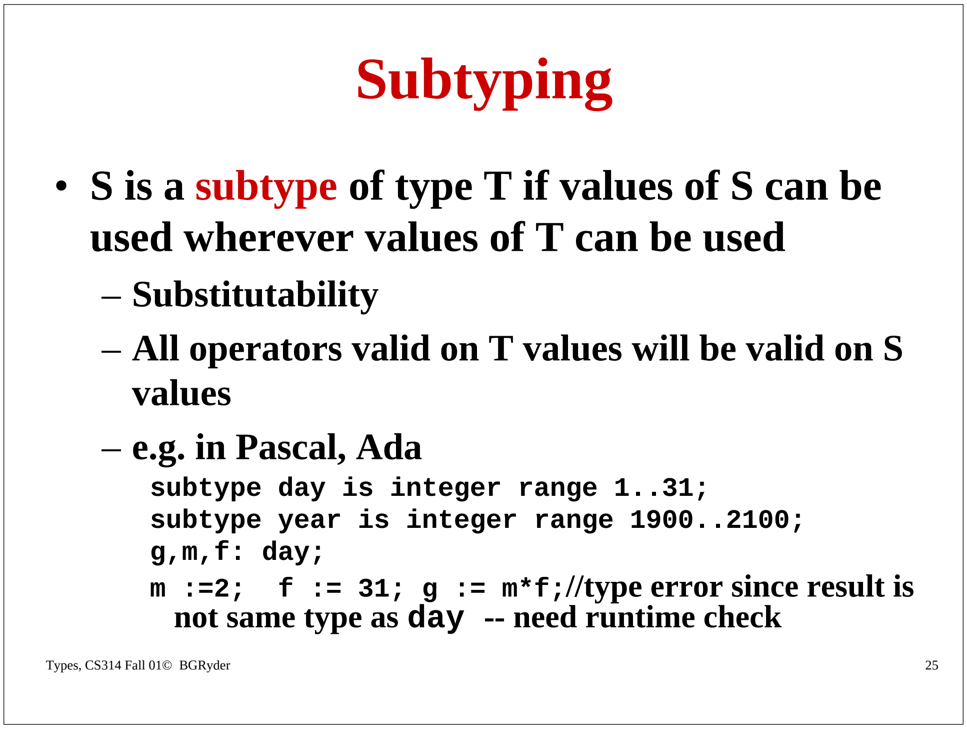
in Ada:

```
type class is (frosh,soph,jr,sr);            //type declaration
stud_class: class;                           //variable declaration
subtype upperclass is class range jr..sr;// subtype decl
joe: upperclass;                             //variable declaration
jr < sr                                      //comparison
for student := frosh.. sr do {…}//use enum type as loop index
college: array[frosh .. sr] of integer;//use enum type as bounds
    range
class ‘pos(soph) is 2; class ‘val(3) is jr //translate enum value
  into position in partial order and vice versa
```

Problems with Enumerations

- Same named literals in 2 types where one is not a subtype of another

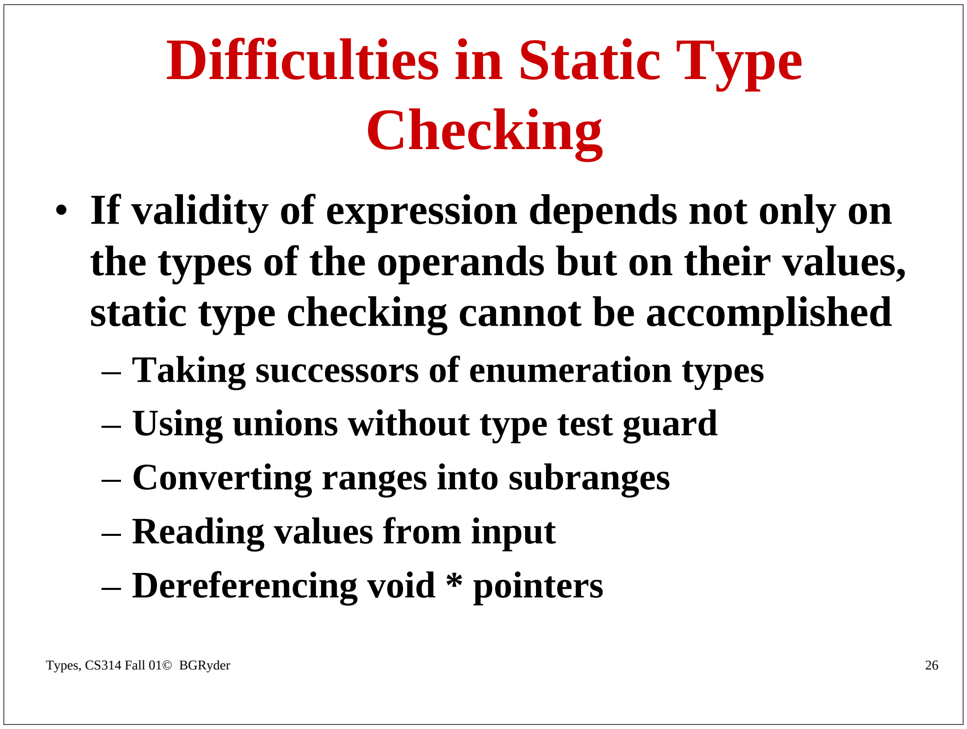
```
type class is (frosh, soph, jr, sr);
type transfer is (jr, sr, grad);
transfer’succ(sr) is grad
class’succ(sr) is UNDEFINED
```



Subtyping

- S is a subtype of type T if values of S can be used wherever values of T can be used
  - Substitutability
  - All operators valid on T values will be valid on S values
  - e.g. in Pascal, Ada

```
subtype day is integer range 1..31;
subtype year is integer range 1900..2100;
g,m,f: day;
m :=2;  f := 31; g := m*f;//type error since result is
  not same type as day  -- need runtime check
```



Difficulties in Static Type Checking

- If validity of expression depends not only on the types of the operands but on their values, static type checking cannot be accomplished
  - Taking successors of enumeration types
  - Using unions without type test guard
  - Converting ranges into subranges
  - Reading values from input
  - Dereferencing void * pointers