

## Announcements

- **Prolog programming due May 3**
  - don't leave this until the last minute
  - at least try to write `rect` before recitation
- **Review session Friday May 7, 12:00-3:00 PM, SEC 218 (with Prof Borgida and Prof Ryder)**
  - a practice final is on the web site
- **Final exam: May 10<sup>th</sup>, 4:00-7:00 pm in the College Ave Gym Annex**
  - let me know about any conflicts by Thursday

1

## 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?

2

# 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**

3

# 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**

4

# Arrays

- **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**
- **This is called a “dope vector”; it 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**

5

# 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**

6

## Strings as a Data Type

- **Length**
  - Can be declared with a maximum length
  - Can have unlimited length
  - Usually allow lexicographic comparison
- **Operations allow string decomposition into substrings and combination**
  - `"abc" + "de" → "abcde"`      *concatenation*
  - `index("abc", "xyabcd") → 2`      *find substring*

7

## 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
- **C: string is sequence of zero or more *char*'s followed by a '`\0`' character**
  - Length is number of *char*'s contained (`strlen`)
  - `strcpy()`, `strcat()`, etc.

8

# 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=NULL;
t = s;
```

9

# 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
```

10

# Type Equivalence

## C example

```
typedef int dollars;
typedef int francs;
dollars x,w;
francs y;
int k;
x=2; w=x;
y=w; y+=k;

typedef struct {int a; int b;} record1;
typedef struct {int b; int a;} record2;
```

11

# Type Equivalence

- **Structural Equivalence defined by the following three rules: (Sethi, p. 140)**
  - SE1: a type name is structurally equivalent to itself.
  - SE2: two types are structurally equivalent if they are formed by applying the same type constructor to two structurally equivalent types.
  - SE3: after type declaration `type n = T`, the type name *n* is structurally equivalent to *T*.
- **Intuitively, user defined types are equivalent under this definition if they have the same “structure” or “shape”**

12

# Type Equivalence

- **Structural Equivalence:**

```
type S = array [0..99] of char;  
type T = array [0..99] of char;
```

```
typedef struct{  
    int j, int k, int *ptr;  
} cell;  
typedef struct{  
    int n, int m, int *p;  
} element;
```

```
typedef int dollars;  
typedef int francs;
```

13

# Type Equivalence

- **Name Equivalence (Sethi, p. 140):**

**(more restricted than structural equivalence)**

- **Pure name equivalence:** A type name is equivalent to itself, but no constructed type is equivalent to any other constructed type.
- **Transitive name equivalence:** A type name is equivalent to itself, and can be declared equivalent to other type names.
- **Type expression equivalence:** A type name is equivalent only to itself. Two type expressions are equivalent if they are formed by applying the same constructor to equivalent expressions (pointers, arrays, templates)

14

# Unions

- **Example (in C):**

```
union mixed {  
    char  c;  
    int   i;  
    float f;  
} value;
```

- **Problem?**

```
value.f = 3.1416;  
...  
printf("Value = %d\n", value.i);
```

15

## Problems with Unions

- **What is meaning of assignment to tag field without assignment to variant fields?**
  - **Ada: must change both value and tag together**
- **If tag not kept in record itself (Pascal), how can its value be checked?**
- **Should tag fields be required to be initialized?**
- **Component selection has to be runtime checked**

16

## Polymorphism (again)

- How to escape from the type system when it becomes a straitjacket?
  - Don't use unions
  - Use *ad hoc* polymorphism (operator overloading in C++)
  - Use subtype polymorphism
    - subranges  
`type Ages = 1..120; type TeenAges = 13..19;`
    - subclasses
  - Use parametric polymorphism (“templates” in C++)
  - Use implicit typing and type “reconstruction” (in ML)

17

## Subtype polymorphism

- 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
```

18

# Parametric Polymorphism

- **Parametrized Types in C++:**

```
template<class T>
class Stack {
    int top;
    int size;
    T *elements;
public:
    Stack(int n) {size=n;elements=new T[size];top=0;}
    ~Stack()      {delete elements;}
    void push(T a);
    T    pop();
};
```

19

# Parametric Polymorphism

- **Parametrized Types in C++:**

```
template<class X>
void Stack<X>::push(X a) {
    top++;
    elements[top] = a;
}
```

- **To use this template, declare:**

```
Stack<int>  s(99);
Stack<char> t(80);
```

- **Thus, we do not have to write separate definitions:**

```
intStack, charStack, ...
```

20