Array Types

Index types can be of any discrete type

Component type must be definite, i.e. have bounds:

```
type class_list is array (1 .. 100) of String (1..10);  -- OK

type class_list is array (1 .. 100) of String;            -- Error
```

The subtype constrains all indices or none:

```
type Matrix is array
    (positive range <> , positive range <> ) of Long_Float;

subtype Table is Matrix;

subtype Rotation is Matrix (1 .. 3 , 1 .. 3);
```

Arrays are objects with assignment: (unlike C, C++)

```
Table1 := Table2;  -- all components assigned
```
Anonymous Array Types

Grades : array (1 .. Num_Students) of Natural;

type of Grades has no name: distinct from any other array types.

Ar1: array (1 .. 10) of Boolean;
Ar2 : array (1 .. 10) of Boolean;
...
Ar1 := Ar2; -- Error: different (anonymous) types.

If a type is a useful abstraction, it deserves to have a name!
**Array Attributes**

type Matrix is array (Positive range <> , Positive range <> )
of Float;

subtype Rect is Matrix (1 .. 3, 1 .. 5);

M3 : Rect;

M3’First (1) -- Yields 1
M3’First -- same.
Rect’length (2) -- Yields 5 (applies to type)
M3’range (2) -- equivalent to 1..5
String’Length -- ERROR: unconstrained

Arrays are self-describing: size information is built-in
Array Aggregates

Expression that yields an array value:

A := (1, 2, 3, 10);  -- positional
A := (1, others => 0);  -- notation for default.
A := (1..3 => 1, 4 => -999);  -- component associations

Default can only be used if bounds are known:

A : String (1 .. 10) := (others => '?');  -- OK
A : String := (others => '?');  -- Error: unknown bounds.
Initializers in C++

- **Similar notion for declarations:**
  
  ```
  int v2[] = {1, 2, 3, 4};   -- size from initializer
  char v3[2] = {'a', 'z'};  -- declared size
  int v5[10] = {-1};         -- default: other components = 0
  char name [ ] = "Algol"   -- String literals are aggregates
  ```

- but no array assignments, so initializer is not an expression (mechanism is less orthogonal)
Aggregates and Qualification

Aggregate may be ambiguous:

```plaintext
type Vector is array (1 .. 3) of Float;
procedure Display (V : vector);
type Assay is array (1 .. 3) of Float;
procedure Display (A : assay);

... Display ((1.0, 1.2, 1.5)); -- which? ambiguous
Display (Vector '(1.0, 1.2, 1.5)); -- OK.
```
Multidimensional Arrays

Aggregates given in row-major order with subaggregates:

```pascal
  type Square is array (1 .. 3, 1 .. 3) of Integer;
  Unit : constant Square := ( (1, 0, 0), (0, 1, 0), (0, 0, 1));
```

A two-dimensional array is NOT an array of arrays:

```pascal
  type vector is array (1 .. 3) of Integer;
  type V3 is array (1 .. 3) of vector;
  -- not convertible to Square
```
Boolean operations extend pointwise:

```plaintext
type Set is array (1 .. Card) of Boolean;
S1, S2, S3 : Set;
...
S3 := S1 and S2;   -- Set Intersection
```

**lexicographic comparisons on arrays of discrete types:**

```plaintext
S1 := (T, T, T);
S2 := (T, T, F);
if ( S2 < S1 ) then ..   -- yields True
```
Both operations yield the base type:

```fortran
  type Table is array (1..10) of Integer;
  T1, T2 : Table;

  ...

  T1 & T2  -- What type?
```

**Declaration equivalent to:**

```fortran
  type Anon is array (integer range <>) of Integer;
  subtype Table is Anon (1 .. 10);
  T1 & T2 , T1 (X .. Y) are of type Anon
```
subsection Sub is Positive range 2 .. 4;
Label : String (1..10) := “transcends” ;
...
Label (2 .. 4) -- Yields “ran”
Label (Integer range 2 .. 4) -- Same
Label (Sub) -- Ditto
Records

type city is record -- Ada
    Name: String (1..10);
    Country: String (1..20);
    Population: integer;
    Capital: Boolean;
end record;

struct city { -- C, C++
    char* name;
    char* country;
    int population
    bool capital }

Variants

type figure_kind is (Circle, Square, Line);
type Figure (Kind : Figure_kind) is record
  Color : color_type;                  -- defined elsewhere
  Visible : Boolean;
  case Kind is
    when Line =>
      Length  : Integer;
      Orientation : Float;
      Start      : Point;              -- defined elsewhere
    when square =>
      Lower_Left, Upper_Right : Point;
    when circle   =>
      Radius   : Integer;
      Center   : Point;
  end case;
end record;
Variants are type safe

C1 : Figure (Circle);  -- discriminant provides constraint
S1 : Figure (Square);
...
C1. Radius := 15;
if S1.Lower_Left = C1.Center then..
function Area (F : Figure) return Float is
   -- applies to any figure, i.e. subtype
begin
   case F.Kind is
      when Circle => return Pi * Radius ** 2;
   ..
Discriminant checking

L : Figure (Line);
F : Figure;                              -- illegal, don’t know which kind
P1, P2 := Point;
...
C := (Circle, Red, False, 10, P1);        -- record aggregate
if C.Orientation then                     -- illegal, circles have no orientation
  C := L;                                 -- illegal, different kinds
  C.Kind := Square;                       -- Illegal, discriminant is constant

Discriminant is visible constant component of object
There is a way of specifying a figure that can change kinds
Variants and classes

- Discriminated types and classes have similar functionalities
- Discriminated types can be allocated statically
- Run-time code uses less indirection
- Compiler can enforce consistent use of discriminants
- Adding new variants is disruptive
  - must modify every case statement
- Variant programming: one **procedure** at a time
- Class programming: one **class** at a time
Free Unions

Free unions can be used to bypass the type model:

```c
union Value {
    char* s;             // allocated at same address (C semantics)
    int i; }
```

Programmer must keep track of current type, e.g. by using an explicit tag:

```c
struct Entry {
    int discr;
    union {              // anonymous component, either s or i.
        char* s;       // if discr = 0
        int i;         // if discr = 1, but run-time system won’t check
    }
};
```
Discriminated unions and dynamic typing

In dynamically-typed languages, only values have types, not names.

\[ S = 13.45 \quad \text{# a floating-point number} \]
\[ \ldots \]
\[ S = [1, 2, 3, 4] \quad \text{# now } S \text{ is a list} \]

Run-time values are described by discriminated unions. Discriminant denotes type of value.

\[ S = X + Y \quad \text{# arithmetic or concatenation} \]

The Variant type in BASIC has the same property. The Tag in a class object functions like a discriminant
Access Types and pointers

Related (but distinct) notions:
- a value that denotes a memory location
- a dynamic name that can designate different objects
- a mechanism to separate stack and heap allocation

```plaintext
type ptr is access integer;   -- Ada: named type

typedef ptr int*;            -- C, C++
```

- A value of type (access T) designates a value of type T
type Cell; -- an incomplete type
  type Ptr is access Cell; -- an access to it
  type Cell is record -- its full declaration
    value : Integer;
    next, prev : Ptr;
  end record;
List: Ptr := new Cell `(10, null, null);
… -- a list is just a pointer to its first element
List.next := new Cell `(15, null, null);
List.next.prev := List;
Incomplete declarations in C++

```c
struct cell {
    int Value;
    cell* prev;       // legal to mention name
    cell* next;       // before end of declaration
};

struct List;

struct Link {
    link* succ;       // a pointer to it
    List* member_of;  // full definition
};

struct List {
    Link* head:      // mutual references
};
```
Pointers and dereferencing

Need notation to distinguish pointer from designated object

- In Ada: *Ptr, Ptr.all
- In C: *Ptr, Ptr*
- In Java: no notion of pointer

For pointers to composite values, dereference can be implicit:

- In Ada: C1.Value equivalent to C1.all.Value
- In C++: distinguish C1.Value and C1 -> Value
- In both: pointers to arrays are indexable: arr_ptr(5), arr_ptr[5]
Four models for arrays

**Ada**, arrays can be static or dynamic. Arrays are objects with assignment. **There are named array types**

In **C++** arrays can be static only if they have static bounds. There is no array assignment.

In **Java** arrays are always dynamic, assignment is a reference assignment. No true multidimensional arrays

In **C#** arrays are always dynamic, fully multidimensional
Arrays in C#

- As in C, C++, Java, describes structure of object, not a named type
- As in Java, always dynamically allocated.
  ```csharp
  int a[ ] = new int[ ]{1, 2, 3, 5, 7, 11, 13, 17, 19};
  ```
- Rectangular arrays:
  ```csharp
  double tab[ , ] =
  { { cos(x), sin(x), 0.0}, {-sin(x), cos(x), 0.0}, {0.0, 0.0, 1.0}};
  ```
- Jagged arrays, or arrays of arrays:
  ```csharp
  int values[ ] [ ] = {{2, 3, 5}, {0, 0}};
  ```
- Orthogonality:
  ```csharp
  int [ , ] [ ] tables := new int[20,20][ ];
  ```
Variations on Strings: Ada

Strings are arrays:

type String is array (positive range <>) of character;
type Str_Ptr is access String;
Ptr1, Ptr2 : Str_Ptr; -- initially null
Title : String := "Brave New World" ; -- fixed size
Ptr3 : Str_Ptr := new String'("Island");

...

Ptr1 := Ptr3; -- pointer assignment makes synonyms
Ptr1.all := "what??"; -- array assignment: must be same size
Ptr1 := new String ("the genius and the goddess");
Title := Ptr1.all; -- run time error: sizes don’t match
Variations on Strings: C++

char* name1, name2;
char title[] = "brave new world";  // 16 characters: implicit 0 at end
const char* t = "island"          // pointer to constant array
name1 = new char[16];             // allocate dynamic storage
const char* ptr = &title[0];      // pointer to local constant array

...  
while (*name1++ == *ptr++);     // amusing C idiom
name1[0] = 'B';                 // title not affected
    t[0] = "I";                  // illegal: string literal is constant

semantic equivalence: a[k] = *(a + k)
Variations on Strings: Java

Strings are classes, not arrays: need special notation for indexing and slicing (charAt...)

String values are constants: need to use arrays of characters to modify strings.

```java
String name = "Eyeless in Gaza";
...

name = name + "(" + 1939 + ");  // assign different value
  // implicit conversion to string: "Eyeless in Gaza (1939)"
if (name.charAt(0) == 'E') {  // true
```
Variations on Strings: C#

- **String** is a class with special syntactic properties:
  - Indexing is available: `s[i]` is a Unicode character
  - Special methods for modification: Remove, Replace, etc.
  - Equality is not identity but equality of contents
  - “+” overloaded as in Java to produce string representations of arbitrary values
Pointers and safety

Pointers create aliases: accessing the value through one name affects the retrieval through the other:

```cpp
int* tab1, tab2;
...
```
debute (tab1);  // discard storage
```
// allocate
// share
// error, tab2 does not denote anything
```
Dangling references

If we can point to local storage, we can create a reference to an undefined value:

```c
int* f() { // returns a pointer to an integer
    int local; // variable on stack frame of f
    ...
    return local&; // reference to local entity
};
int x = f();
... x + 1 ... // stack may have been overwritten
```
because not all current references to an object may be visible. System is safer if storage reclamation is automated.

- **Manual solution**: make deallocation more explicit:
  
  ```plaintext
  procedure free is new Ada.Unchecked_Deallocation (String, Ptr);
  ```

- **Semi-automatic solution** (Ada, C++): destructors, controlled types

- **Automatic Solution** (Java, ML): garbage collector