



HINDUSTAN

INSTITUTE OF TECHNOLOGY & SCIENCE
(DEEMED TO BE UNIVERSITY)



CSB4404 – PROGRAMMING PARADIGMS

B.Tech – VII Semester
Lecture 10

Dr. Muthukumaran M
Associate professor
School of Computing Sciences,
Department of Computer Science and Engineering

Data Types and Statements

Unit II



Topics Introduction Primitive Data Types

- Character String Types
- User-Defined Ordinal Types
- Array Types
- Associative Arrays
- Record Types
- Tuple Types
- List Types
- Union Types
- Pointer and Reference Types
- Type Checking
- Strong Typing
- Type Equivalence
- Theory and Data Types

Introduction

- A data type defines a collection of data objects and a set of predefined operations on those objects
- A descriptor is the collection of the attributes of a variable
- An object represents an instance of a user-defined (abstract data) type
- One design issue for all data types: What operations are defined and how are they specified?

Primitive Data Types

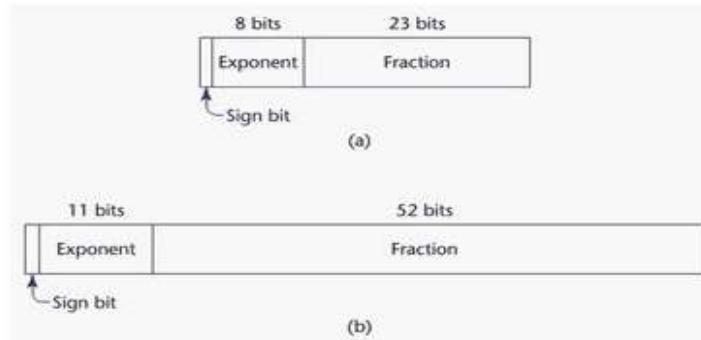
- Almost all programming languages provide a set of primitive data types
- Primitive data types: Those not defined in terms of other data types
- Some primitive data types are merely reflections of the hardware
- Others require only a little non-hardware support for their implementation

Primitive Data Types: Integer

- Almost always an exact reflection of the hardware so the mapping is trivial
- There may be as many as eight different integer types in a language
- Java's signed integer sizes: byte, short, int, long

Primitive Data Types: Floating Point

- Model real numbers, but only as approximations
- Languages for scientific use support at least two floating-point types (e.g., `float` and `double`; sometimes more)
- Usually exactly like the hardware, but not always
- IEEE Floating-Point Standard 754



Primitive Data Types: Complex

- Some languages support a complex type, e.g., C99, Fortran, and Python
- Each value consists of two floats, the real part and the imaginary part
- Literal form (in Python):
 $(7 + 3j)$, where 7 is the real part and 3 is the imaginary part

Primitive Data Types: Decimal

- For business applications (money)
 - Essential to COBOL
 - C# offers a decimal data type
- Store a fixed number of decimal digits, in coded form (BCD)
- *Advantage:* accuracy
- *Disadvantages:* limited range, wastes memory

Primitive Data Types: Boolean

- Simplest of all
- Range of values: two elements, one for “true” and one for “false”
- Could be implemented as bits, but often as bytes
 - Advantage: readability

Primitive Data Types: Character

- Stored as numeric codings
- Most commonly used coding: ASCII
- An alternative, 16-bit coding: Unicode (UCS-2)
 - Includes characters from most natural languages
 - Originally used in Java
 - C# and JavaScript also support Unicode
- 32-bit Unicode (UCS-4)
 - Supported by Fortran, starting with 2003

Character String Types

- Values are sequences of characters
- Design issues:
 - Is it a primitive type or just a special kind of array?
 - Should the length of strings be static or dynamic?

Character String Types Operations

- Typical operations:
 - Assignment and copying
 - Comparison ($=$, $>$, etc.)
 - Catenation
 - Substring reference
 - Pattern matching

Character String Type in Certain Languages

- C and C++
 - Not primitive
 - Use `char` arrays and a library of functions that provide operations
- SNOBOL4 (a string manipulation language)
 - Primitive
 - Many operations, including elaborate pattern matching
- Fortran and Python
 - Primitive type with assignment and several operations
- Java
 - Primitive via the `String` class
- Perl, JavaScript, Ruby, and PHP
 - Provide built-in pattern matching, using regular expressions

Character String Length Options

- Static: COBOL, Java's String **class**
- *Limited Dynamic Length*: C and C++
 - In these languages, a special character is used to indicate the end of a string's characters, rather than maintaining the length
- *Dynamic* (no maximum): SNOBOL4, Perl, JavaScript
- Ada supports all three string length options

Character String Type Evaluation

- Aid to writability
- As a primitive type with static length, they are inexpensive to provide--why not have them?
- Dynamic length is nice, but is it worth the expense?

Character String Implementation

- Static length: compile-time descriptor
- Limited dynamic length: may need a run-time descriptor for length (but not in C and C++)
- Dynamic length: need run-time descriptor; allocation/deallocation is the biggest implementation problem

Compile- and Run-Time Descriptors

Static string
Length
Address

**Compile-time
descriptor for
static strings**

Limited dynamic string
Maximum length
Current length
Address

**Run-time
descriptor for
limited dynamic
strings**

User-Defined Ordinal Types

- An ordinal type is one in which the range of possible values can be easily associated with the set of positive integers
- Examples of primitive ordinal types in Java
 - `integer`
 - `char`
 - `boolean`

Enumeration Types

- All possible values, which are named constants, are provided in the definition
- C# example

```
enum days {mon, tue, wed, thu, fri, sat, sun};
```
- Design issues
 - Is an enumeration constant allowed to appear in more than one type definition, and if so, how is the type of an occurrence of that constant checked?
 - Are enumeration values coerced to integer?
 - Any other type coerced to an enumeration type?

Evaluation of Enumerated Type

- Aid to readability, e.g., no need to code a color as a number
- Aid to reliability, e.g., compiler can check:
 - operations (don't allow colors to be added)
 - No enumeration variable can be assigned a value outside its defined range
 - Ada, C#, and Java 5.0 provide better support for enumeration than C++ because enumeration type variables in these languages are not coerced into integer types

Subrange Types

- An ordered contiguous subsequence of an ordinal type
 - Example: 12..18 is a subrange of integer type
- Ada's design

```
type Days is (mon, tue, wed, thu, fri, sat, sun);
subtype Weekdays is Days range mon..fri;
subtype Index is Integer range 1..100;
```

Day1: Days;

Day2: Weekday;

Day2 := Day1;

Department of Computer science and Engineering

CSB4404 – PROGRAMMING PARADIGMS

22

Subrange Evaluation

- Aid to readability
 - Make it clear to the readers that variables of subrange can store only certain range of values
- Reliability
 - Assigning a value to a subrange variable that is outside the specified range is detected as an error

Implementation of User-Defined Ordinal Types

- Enumeration types are implemented as integers
- Subrange types are implemented like the parent types with code inserted (by the compiler) to restrict assignments to subrange variables

Array Types

- An array is a homogeneous aggregate of data elements in which an individual element is identified by its position in the aggregate, relative to the first element.

Array Design Issues

- What types are legal for subscripts?
- Are subscripting expressions in element references range checked?
- When are subscript ranges bound?
- When does allocation take place?
- Are ragged or rectangular multidimensional arrays allowed, or both?
- What is the maximum number of subscripts?
- Can array objects be initialized?
- Are any kind of slices supported?

Array Indexing

- *Indexing* (or subscripting) is a mapping from indices to elements
array_name (index_value_list) → an element
- **Index Syntax**
 - Fortran and Ada use parentheses
 - Ada explicitly uses parentheses to show uniformity between array references and function calls because both are *mappings*
 - Most other languages use brackets

Arrays Index (Subscript) Types

- FORTRAN, C: integer only
- Ada: integer or enumeration (includes Boolean and char)
- Java: integer types only
- Index range checking
 - C, C++, Perl, and Fortran do not specify range checking
 - Java, ML, C# specify range checking
 - In Ada, the default is to require range checking, but it can be turned off

Subscript Binding and Array Categories

- *Static*: subscript ranges are statically bound and storage allocation is static (before run-time)
 - Advantage: efficiency (no dynamic allocation)
- *Fixed stack-dynamic*: subscript ranges are statically bound, but the allocation is done at declaration time
 - Advantage: space efficiency

Subscript Binding and Array Categories (continued)

- *Stack-dynamic*: subscript ranges are dynamically bound and the storage allocation is dynamic (done at run-time)
 - Advantage: flexibility (the size of an array need not be known until the array is to be used)
- *Fixed heap-dynamic*: similar to fixed stack-dynamic: storage binding is dynamic but fixed after allocation (i.e., binding is done when requested and storage is allocated from heap, not stack)

Subscript Binding and Array Categories (continued)

- Heap-dynamic: binding of subscript ranges and storage allocation is dynamic and can change any number of times
 - Advantage: flexibility (arrays can grow or shrink during program execution)

Subscript Binding and Array Categories (continued)

- C and C++ arrays that include `static` modifier are static
- C and C++ arrays without `static` modifier are fixed stack-dynamic
- C and C++ provide fixed heap-dynamic arrays
- C# includes a second array class `ArrayList` that provides fixed heap-dynamic
- Perl, JavaScript, Python, and Ruby support heap-dynamic arrays

Array Initialization

- Some language allow initialization at the time of storage allocation

- C, C++, Java, C# example

```
int list [] = {4, 5, 7, 83}
```

- Character strings in C and C++

```
char name [] = "freddie";
```

- Arrays of strings in C and C++

```
char *names [] = {"Bob", "Jake", "Joe"};
```

- Java initialization of String objects

```
String[] names = {"Bob", "Jake", "Joe"};
```

Heterogeneous Arrays

- A *heterogeneous array* is one in which the elements need not be of the same type
- Supported by Perl, Python, JavaScript, and Ruby

Array Initialization

- C-based languages
 - `int list [] = {1, 3, 5, 7}`
 - `char *names [] = {"Mike", "Fred", "Mary Lou"};`
- Ada
 - `List : array (1..5) of Integer :=
(1 => 17, 3 => 34, others => 0);`
- Python
 - List comprehensions

```
list = [x ** 2 for x in range(12) if x % 3 == 0]  
puts [0, 9, 36, 81] in list
```

Arrays Operations

- APL provides the most powerful array processing operations for vectors and matrixes as well as unary operators (for example, to reverse column elements)
- Ada allows array assignment but also catenation
- Python's array assignments, but they are only reference changes. Python also supports array catenation and element membership operations
- Ruby also provides array catenation
- Fortran provides *elemental* operations because they are between pairs of array elements
 - For example, + operator between two arrays results in an array of the sums of the element pairs of the two arrays

Rectangular and Jagged Arrays

- A rectangular array is a multi-dimensioned array in which all of the rows have the same number of elements and all columns have the same number of elements
- A jagged matrix has rows with varying number of elements
 - Possible when multi-dimensioned arrays actually appear as arrays of arrays
- C, C++, and Java support jagged arrays
- Fortran, Ada, and C# support rectangular arrays (C# also supports jagged arrays)

Slices

- A slice is some substructure of an array; nothing more than a referencing mechanism
- Slices are only useful in languages that have array operations

Slice Examples

- **Python**

```
vector = [2, 4, 6, 8, 10, 12, 14, 16]
```

```
mat = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

vector (3:6) is a three-element array

mat[0][0:2] is the first and second element of the
first row of mat

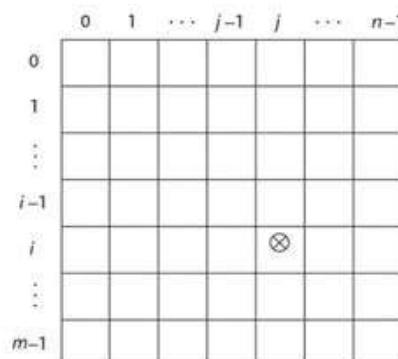
- **Ruby supports slices with the slice method**

list.slice(2, 2) returns the third and fourth
elements of list

Implementation of Arrays

- Access function maps subscript expressions to an address in the array
- Access function for single-dimensioned arrays:

$$\text{address}(\text{list}[k]) = \text{address}(\text{list}[\text{lower_bound}]) + ((k - \text{lower_bound}) * \text{element_size})$$



Accessing Multi-dimensioned Arrays

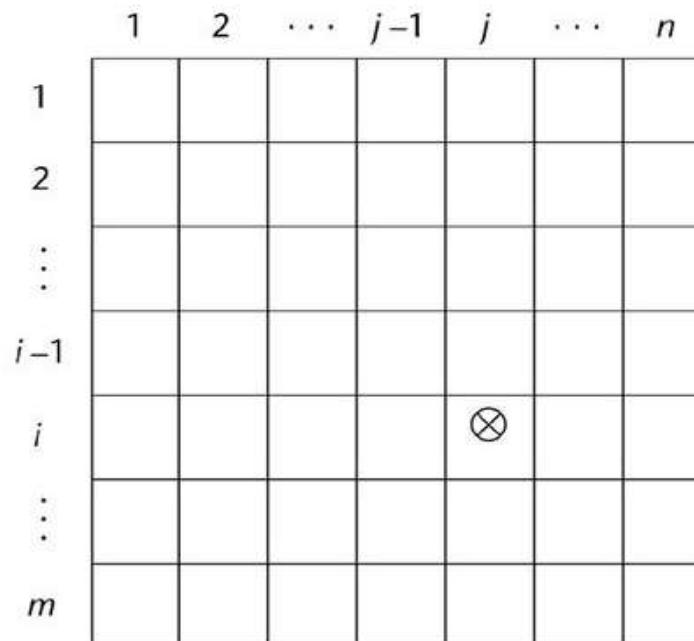
- Two common ways:
 - Row major order (by rows) – used in most languages
 - Column major order (by columns) – used in Fortran
 - A compile-time descriptor for a multidimensional array

Multidimensioned array
Element type
Index type
Number of dimensions
Index range 0
:
Index range $n - 1$
Address

Locating an Element in a Multi-dimensioned Array

- General format

Location ($a[i,j]$) = address of a [row_lb,col_lb] +
 $((i - \text{row_lb}) * n) + (j - \text{col_lb}) * \text{element_size}$



Compile-Time Descriptors

Array
Element type
Index type
Index lower bound
Index upper bound
Address

Single-dimensioned array

Multidimensioned array
Element type
Index type
Number of dimensions
Index range 1
:
Index range n
Address

Multidimensional array

Associative Arrays

- An *associative array* is an unordered collection of data elements that are indexed by an equal number of values called *keys*
 - User-defined keys must be stored
- Design issues:
 - What is the form of references to elements?
 - Is the size static or dynamic?
- Built-in type in Perl, Python, Ruby, and Lua
 - In Lua, they are supported by tables

Associative Arrays in Perl

- Names begin with %; literals are delimited by parentheses

```
%hi_temps = ("Mon" => 77, "Tue" => 79, "Wed" =>  
65, ...);
```

- Subscripting is done using braces and keys

```
$hi_temps{"Wed"} = 83;
```

- Elements can be removed with `delete`

```
delete $hi_temps{"Tue"};
```

Record Types

- A *record* is a possibly heterogeneous aggregate of data elements in which the individual elements are identified by names
- Design issues:
 - What is the syntactic form of references to the field?
 - Are elliptical references allowed

Definition of Records in COBOL

- COBOL uses level numbers to show nested records; others use recursive definition

```
01 EMP-REC.  
    02 EMP-NAME.  
        05 FIRST PIC X(20).  
        05 MID    PIC X(10).  
        05 LAST   PIC X(20).  
    02 HOURLY-RATE PIC 99V99.
```

Definition of Records in Ada

- Record structures are indicated in an orthogonal way

```
type Emp_Rec_Type is record
    First: String (1..20);
    Mid: String (1..10);
    Last: String (1..20);
    Hourly_Rate: Float;
end record;
Emp_Rec: Emp_Rec_Type;
```

References to Records

- Record field references
 1. COBOL
`field_name OF record_name_1 OF ... OF record_name_n`
 2. Others (dot notation)
`record_name_1.record_name_2. ... record_name_n.field_name`
- Fully qualified references must include all record names
- Elliptical references allow leaving out record names as long as the reference is unambiguous, for example in COBOL
FIRST, FIRST OF EMP-NAME, and FIRST of EMP-REC are elliptical references to the employee's first name

Operations on Records

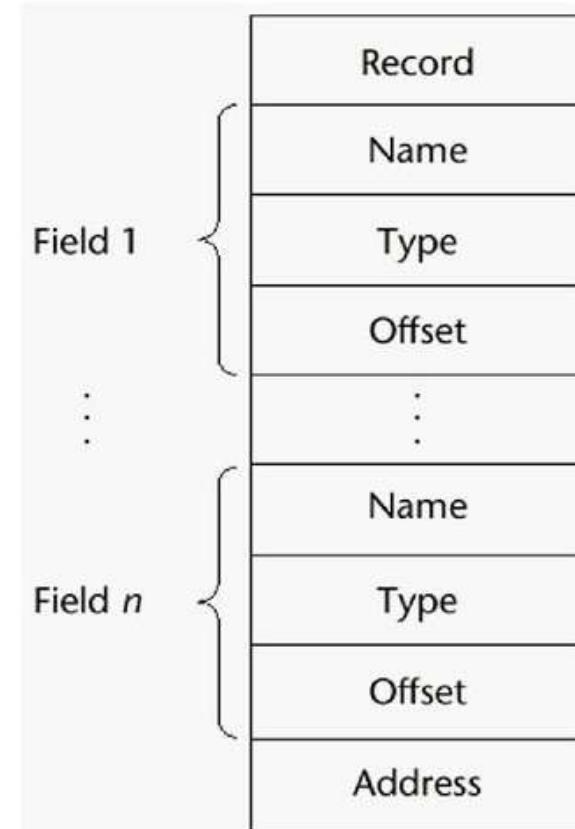
- Assignment is very common if the types are identical
- Ada allows record comparison
- Ada records can be initialized with aggregate literals
- COBOL provides MOVE CORRESPONDING
 - Copies a field of the source record to the corresponding field in the target record

Evaluation and Comparison to Arrays

- Records are used when collection of data values is heterogeneous
- Access to array elements is much slower than access to record fields, because subscripts are dynamic (field names are static)
- Dynamic subscripts could be used with record field access, but it would disallow type checking and it would be much slower

Implementation of Record Type

Offset address relative to the beginning of the records is associated with each field



Tuple Types

- A tuple is a data type that is similar to a record, except that the elements are not named
- Used in Python, ML, and F# to allow functions to return multiple values

– Python

- Closely related to its lists, but immutable
- Create with a tuple literal

```
myTuple = (3, 5.8, 'apple')
```

Referenced with subscripts (begin at 1)

Catenation with + and deleted with del

Tuple Types (continued)

- ML

```
val myTuple = (3, 5.8, 'apple');
```

- Access as follows:

#1(myTuple) is the first element

- A new tuple type can be defined

```
type intReal = int * real;
```

- F#

```
let tup = (3, 5, 7)
```

let a, b, c = tup This assigns a tuple to
a tuple pattern (a, b, c)

List Types

- Lists in LISP and Scheme are delimited by parentheses and use no commas
 $(A\ B\ C\ D)$ and $(A\ (B\ C)\ D)$
- Data and code have the same form
 - As data, $(A\ B\ C)$ is literally what it is
 - As code, $(A\ B\ C)$ is the function A applied to the parameters B and C
- The interpreter needs to know which a list is, so if it is data, we quote it with an apostrophe

$'(A\ B\ C)$ is data

List Types (continued)

- List Operations in Scheme
 - CAR **returns** the first element of its list parameter
`(CAR '(A B C))` **returns** A
 - CDR **returns** the remainder of its list parameter after the first element has been removed
`(CDR '(A B C))` **returns** (B C)
 - CONS **puts** its first parameter into its second parameter, a list, to make a new list
`(CONS 'A (B C))` **returns** (A B C)
 - LIST **returns** a new list of its parameters
`(LIST 'A 'B '(C D))` **returns** (A B (C D))

List Types (continued)

- List Operations in ML
 - Lists are written in brackets and the elements are separated by commas
 - List elements must be of the same type
 - The Scheme `CONS` function is a binary operator in ML, `::`
`3 :: [5, 7, 9]` evaluates to `[3, 5, 7, 9]`
 - The Scheme `CAR` and `CDR` functions are named `hd` and `tl`, respectively

List Types (continued)

- F# Lists
 - Like those of ML, except elements are separated by semicolons and `hd` and `tl` are methods of the `List` class
- Python Lists
 - The list data type also serves as Python's arrays
 - Unlike Scheme, Common LISP, ML, and F#, Python's lists are mutable
 - Elements can be of any type
 - Create a list with an assignment

```
myList = [3, 5.8, "grape"]
```

List Types (continued)

- Python Lists (continued)
 - List elements are referenced with subscripting, with indices beginning at zero

```
x = myList[1] Sets x to 5.8
```
 - List elements can be deleted with `del`

```
del myList[1]
```
 - List Comprehensions – derived from set notation

`[x * x for x in range(6) if x % 3 == 0]`

`range(12) creates [0, 1, 2, 3, 4, 5, 6]`

Constructed list: `[0, 9, 36]`

List Types (continued)

- Haskell's List Comprehensions
 - The original

```
[n * n | n <- [1..10]]
```

- F#'s List Comprehensions

```
let myArray = [|for i in 1 .. 5 -> [i * i)|]
```

- Both C# and Java supports lists through their generic heap-dynamic collection classes, **List** and **ArrayList**, respectively

Unions Types

- A *union* is a type whose variables are allowed to store different type values at different times during execution
- Design issues
 - Should type checking be required?
 - Should unions be embedded in records?

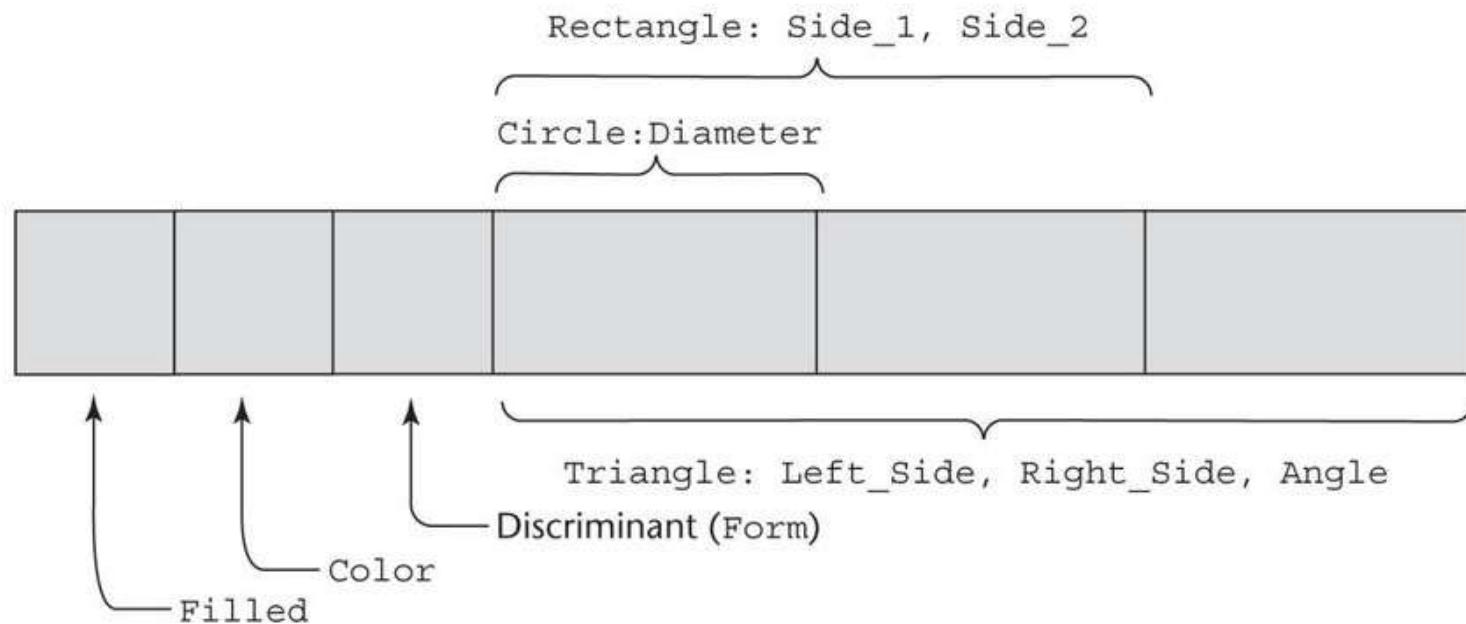
Discriminated vs. Free Unions

- Fortran, C, and C++ provide union constructs in which there is no language support for type checking; the union in these languages is called *free union*
- Type checking of unions require that each union include a type indicator called a *discriminant*
 - Supported by Ada

Ada Union Types

```
type Shape is (Circle, Triangle, Rectangle);
type Colors is (Red, Green, Blue);
type Figure (Form: Shape) is record
    Filled: Boolean;
    Color: Colors;
    case Form is
        when Circle => Diameter: Float;
        when Triangle =>
            Leftside, Rightside: Integer;
            Angle: Float;
        when Rectangle => Side1, Side2: Integer;
    end case;
end record;
```

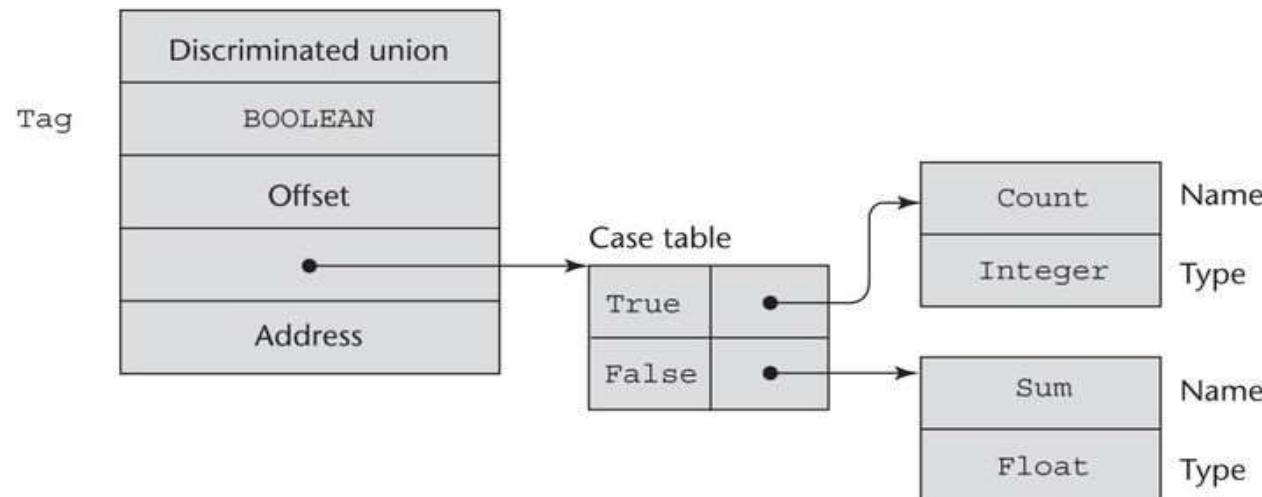
Ada Union Type Illustrated



A discriminated union of three shape variables

Implementation of Unions

```
type Node (Tag : Boolean) is
  record
    case Tag is
      when True => Count : Integer;
      when False => Sum : Float;
    end case;
  end record;
```



Evaluation of Unions

- Free unions are unsafe
 - Do not allow type checking
- Java and C# do not support unions
 - Reflective of growing concerns for safety in programming language
- Ada's discriminated unions are safe