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

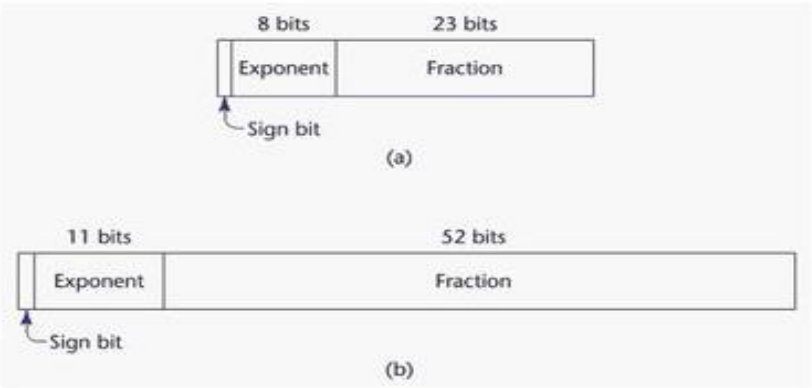
- Almost all programming languages provide a set of primitive data types
- Primitive data types: Those not defined in terms of other data types
- Some primitive data types are merely reflections of the hardware
- Others require only a little non-hardware support for their implementation

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

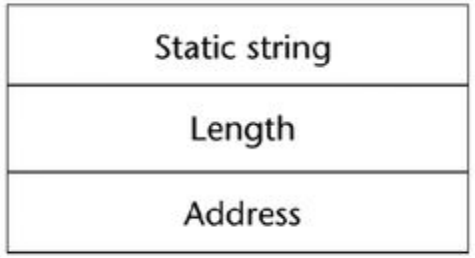
Character String Type Evaluation

- 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?

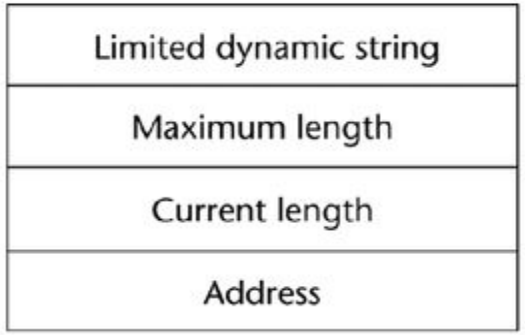
Character String Implementation

- 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



Compile-time descriptor for static strings



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;
```

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`

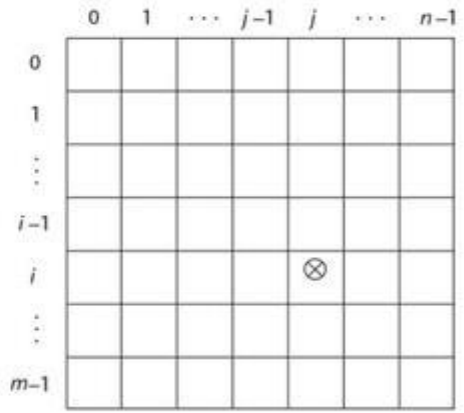
- Ruby supports slices with the `slice` method

`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:

  address(list[k]) = address (list[lower_bound])
  + ((k-lower_bound) * element_size)



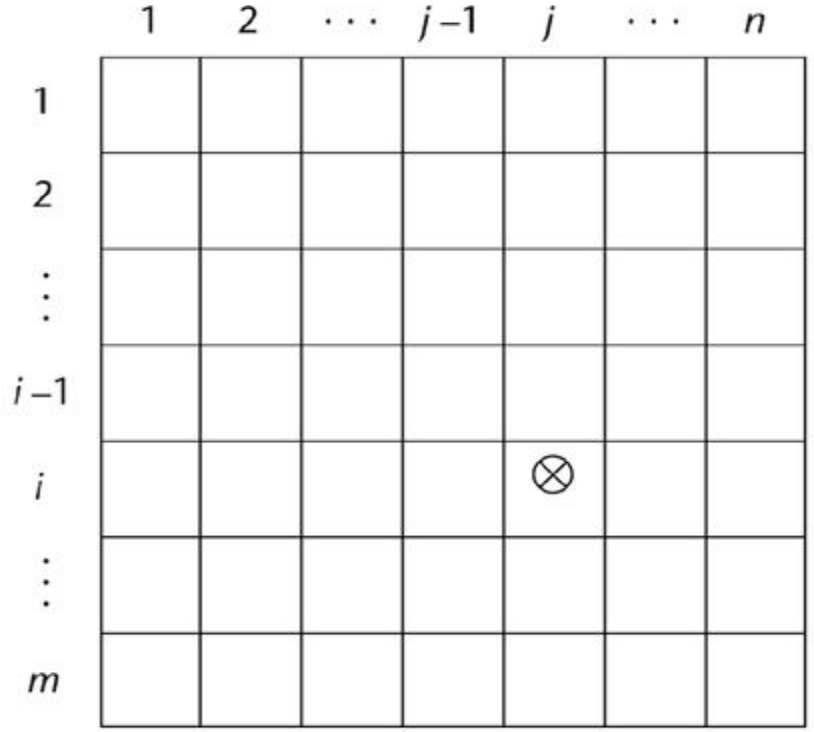
Accessing Multi-dimensioned Arrays

- 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

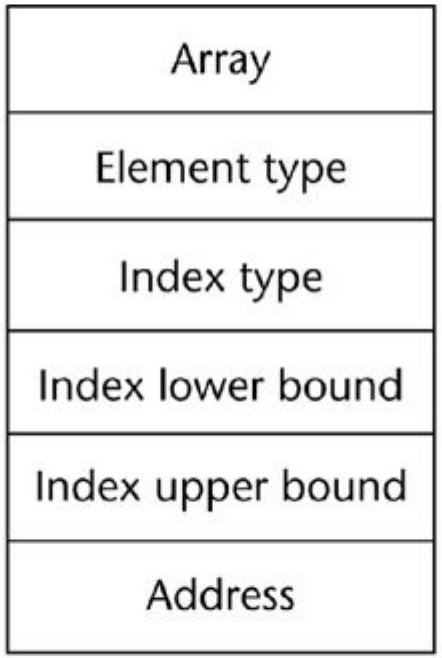
Locating an Element in a Multi-dimensioned Array

- General format

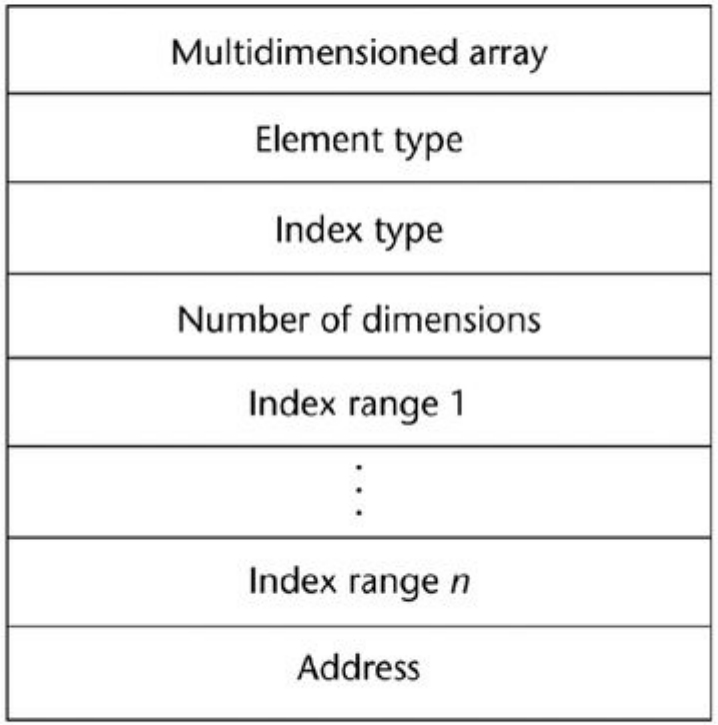
  Location (a[I,j]) = address of a [row_lb,col_lb] +
  (((I – row_lb) * n) + (j – col_lb)) * element_size



Compile-Time Descriptors



Single-dimensioned array



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

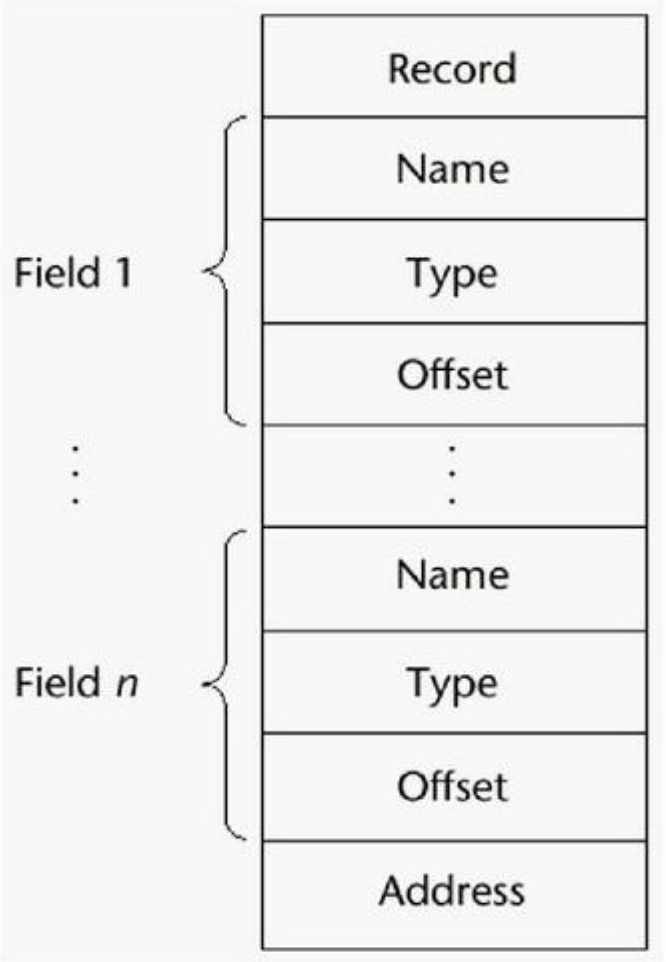
- Assignment is very common if the types are identical
- Ada allows record comparison
- Ada records can be initialized with aggregate literals
- COBOL provides `MOVE CORRESPONDING`
  - Copies a field of the source record to the corresponding field in the target record

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?

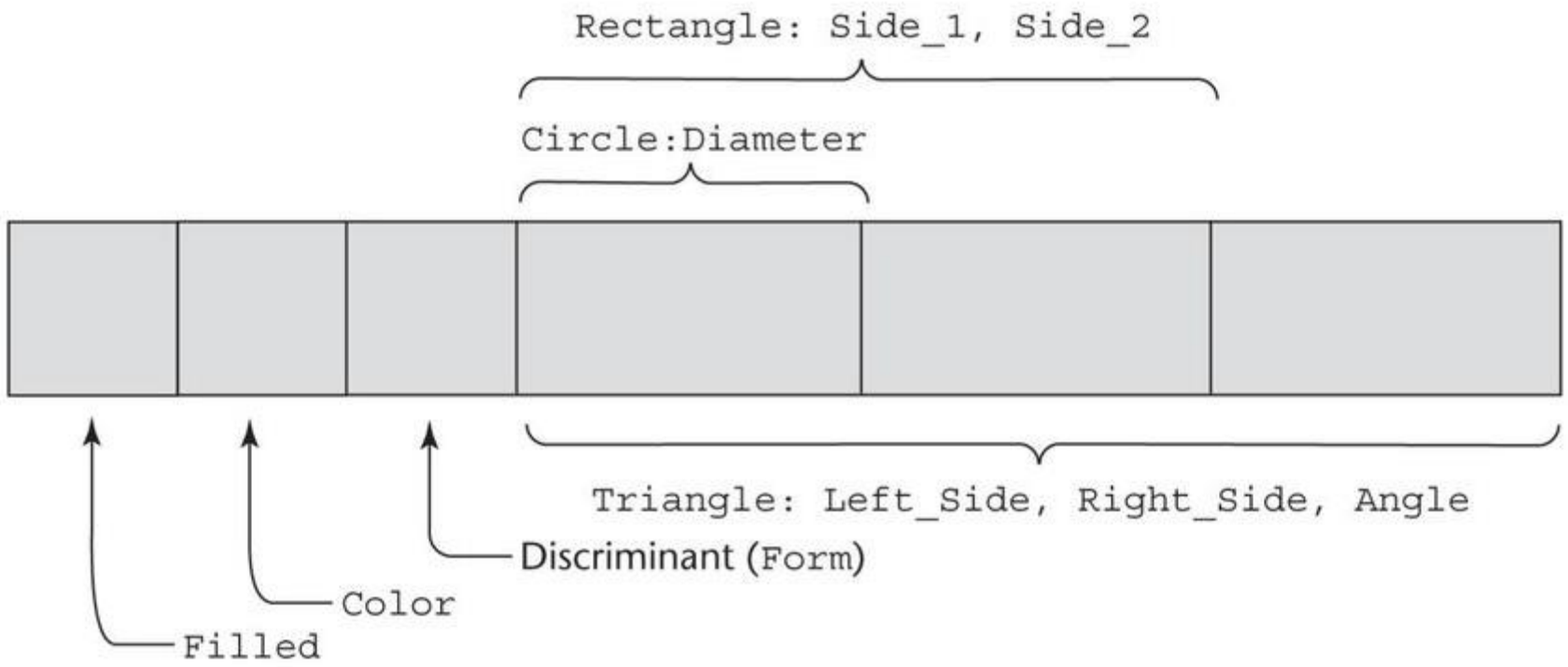
Discriminated vs. Free Unions

- 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;
```

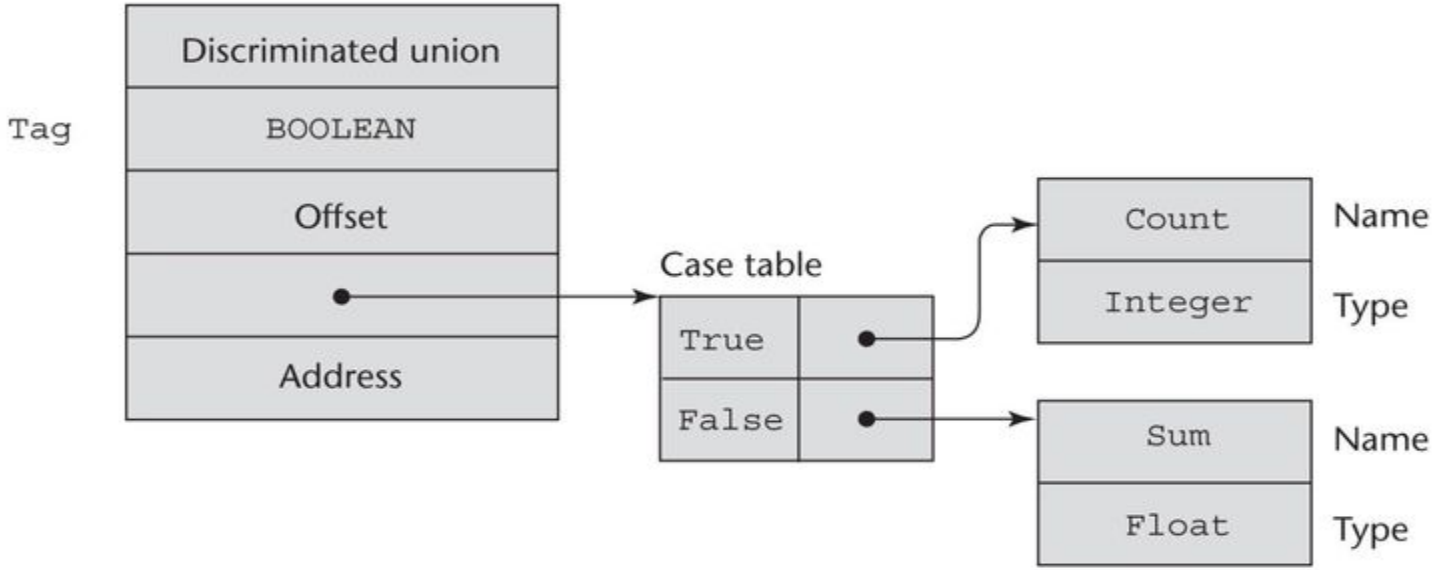
Ada Union Type Illustrated



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 descriminated unions are safe

Pointer and Reference Types

- A *pointer* type variable has a range of values that consists of memory addresses and a special value, *nil*
- Provide the power of indirect addressing
- Provide a way to manage dynamic memory
- A pointer can be used to access a location in the area where storage is dynamically created (usually called a *heap*)

Design Issues of Pointers

- What are the scope of and lifetime of a pointer variable?
- What is the lifetime of a heap-dynamic variable?
- Are pointers restricted as to the type of value to which they can point?
- Are pointers used for dynamic storage management, indirect addressing, or both?
- Should the language support pointer types, reference types, or both?

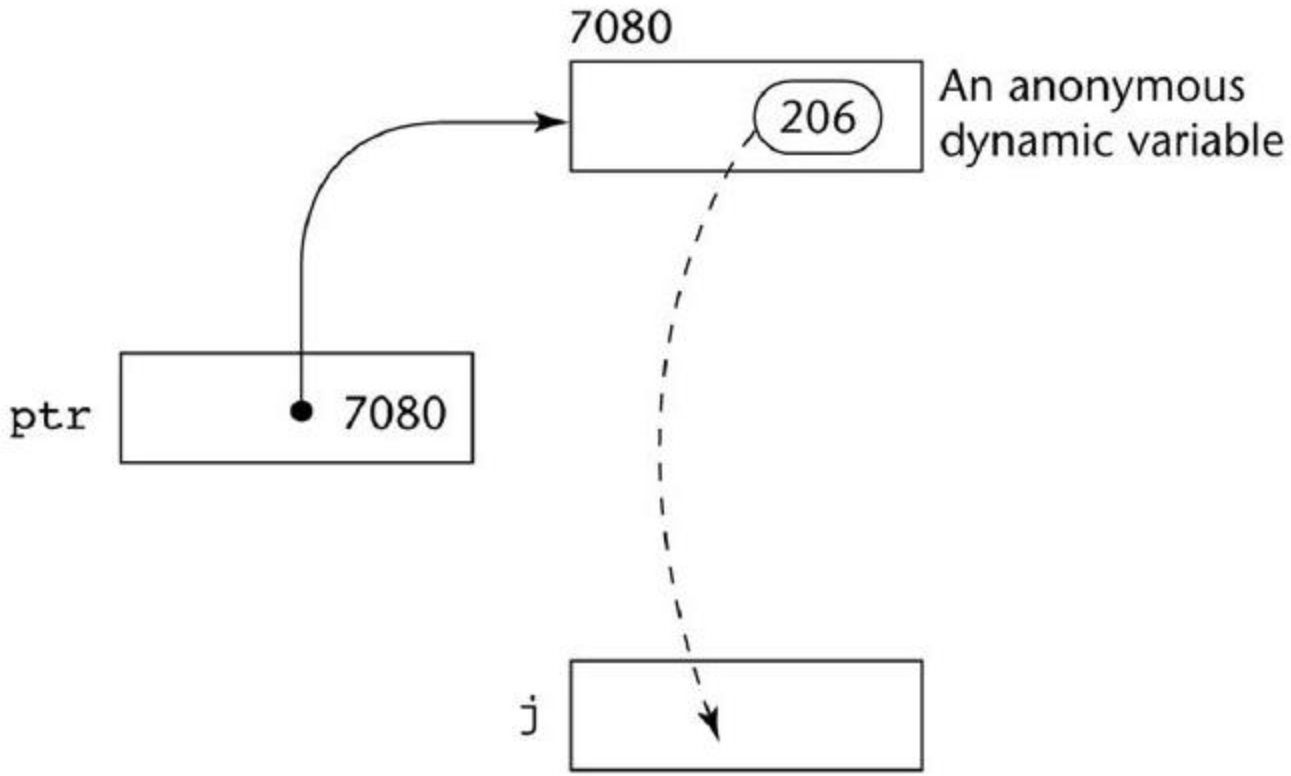
Pointer Operations

- Two fundamental operations: assignment and dereferencing
- Assignment is used to set a pointer variable's value to some useful address
- Dereferencing yields the value stored at the location represented by the pointer's value
  - Dereferencing can be explicit or implicit
  - C++ uses an explicit operation via *

    `j = *ptr`

    sets j to the value located at `ptr`

Pointer Assignment Illustrated



The assignment operation j = *ptr

Problems with Pointers

- Dangling pointers (dangerous)
  - A pointer points to a heap-dynamic variable that has been deallocated
- Lost heap-dynamic variable
  - An allocated heap-dynamic variable that is no longer accessible to the user program (often called *garbage*)
    - Pointer `p1` is set to point to a newly created heap-dynamic variable
    - Pointer `p1` is later set to point to another newly created heap-dynamic variable
    - The process of losing heap-dynamic variables is called *memory leakage*

Pointers in Ada

- Some dangling pointers are disallowed because dynamic objects can be automatically deallocated at the end of pointer's type scope
- The lost heap-dynamic variable problem is not eliminated by Ada (possible with `UNCHECKED_DEALLOCATION`)

Pointers in Ada

- Some dangling pointers are disallowed because dynamic objects can be automatically deallocated at the end of pointer's type scope
- The lost heap-dynamic variable problem is not eliminated by Ada (possible with `UNCHECKED_DEALLOCATION`)

Pointers in C and C++

- Extremely flexible but must be used with care
- Pointers can point at any variable regardless of when or where it was allocated
- Used for dynamic storage management and addressing
- Pointer arithmetic is possible
- Explicit dereferencing and address-of operators
- Domain type need not be fixed (`void *`)

  `void *` can point to any type and can be type checked (cannot be de-referenced)

Pointer Arithmetic in C and C++

```
float stuff[100];
float *p;
p = stuff;
```

`*(p+5)` is equivalent to `stuff[5]` and `p[5]`

`*(p+i)` is equivalent to `stuff[i]` and `p[i]`

Reference Types

- C++ includes a special kind of pointer type called a *reference type* that is used primarily for formal parameters
  - Advantages of both pass-by-reference and pass-by-value
- Java extends C++'s reference variables and allows them to replace pointers entirely
  - References are references to objects, rather than being addresses
- C# includes both the references of Java and the pointers of C++

Evaluation of Pointers

- Dangling pointers and dangling objects are problems as is heap management
- Pointers are like `goto`'s--they widen the range of cells that can be accessed by a variable
- Pointers or references are necessary for dynamic data structures--so we can't design a language without them

Representations of Pointers

- Large computers use single values
- Intel microprocessors use segment and offset

Dangling Pointer Problem

- *Tombstone*: extra heap cell that is a pointer to the heap-dynamic variable
  - The actual pointer variable points only at tombstones
  - When heap-dynamic variable de-allocated, tombstone remains but set to nil
  - Costly in time and space
- *Locks-and-keys*: Pointer values are represented as (key, address) pairs
  - Heap-dynamic variables are represented as variable plus cell for integer lock value
  - When heap-dynamic variable allocated, lock value is created and placed in lock cell and key cell of pointer

Type Checking

- Generalize the concept of operands and operators to include subprograms and assignments
- *Type checking* is the activity of ensuring that the operands of an operator are of compatible types
- A *compatible type* is one that is either legal for the operator, or is allowed under language rules to be implicitly converted, by compiler- generated code, to a legal type
  - This automatic conversion is called a *coercion*.
- A *type error* is the application of an operator to an operand of an inappropriate type

Type Checking (continued)

- If all type bindings are static, nearly all type checking can be static
- If type bindings are dynamic, type checking must be dynamic
- A programming language is *strongly typed* if type errors are always detected
- Advantage of strong typing: allows the detection of the misuses of variables that result in type errors

Strong Typing

Language examples:

- C and C++ are not: parameter type checking can be avoided; unions are not type checked
- Ada is, almost (`UNCHECKED CONVERSION` is loophole)

  (Java and C# are similar to Ada)

Strong Typing (continued)

- Coercion rules strongly affect strong typing--they can weaken it considerably (C++ versus Ada)

- Although Java has just half the assignment coercions of C++, its strong typing is still far less effective than that of Ada

Name Type Equivalence

- *Name type equivalence* means the two variables have equivalent types if they are in either the same declaration or in declarations that use the same type name
- Easy to implement but highly restrictive:
  - Subranges of integer types are not equivalent with integer types
  - Formal parameters must be the same type as their corresponding actual parameters

Structure Type Equivalence

- *Structure type equivalence* means that two variables have equivalent types if their types have identical structures
- More flexible, but harder to implement

Type Equivalence (continued)

- Consider the problem of two structured types:
  - Are two record types equivalent if they are structurally the same but use different field names?
  - Are two array types equivalent if they are the same except that the subscripts are different? (e.g. `[1..10]` and `[0..9]`)
  - Are two enumeration types equivalent if their components are spelled differently?
  - With structural type equivalence, you cannot differentiate between types of the same structure (e.g. different units of speed, both float)

Theory and Data Types

- Type theory is a broad area of study in mathematics, logic, computer science, and philosophy
- Two branches of type theory in computer science:
  - Practical – data types in commercial languages
  - Abstract – typed lambda calculus

- A type system is a set of types and the rules that govern their use in programs

Theory and Data Types (continued)

- Formal model of a type system is a set of types and a collection of functions that define the type rules
  - Either an attribute grammar or a type map could be used for the functions
  - Finite mappings – model arrays and functions
  - Cartesian products – model tuples and records
  - Set unions – model union types
  - Subsets – model subtypes

Expressions and Assignment Statements

- **Introduction**
- ƒExpressions are the fundamental means of specifying computations in a programming language.
- ƒTo understand expression evaluation, need to be familiar with the orders of operator and operand evaluation.
- ƒEssence of imperative languages is dominant role of assignment statements

Arithmetic Expressions

- Their evaluation was one of the motivations for the development of the first programming languages.
- Most of the characteristics of arithmetic expressions in programming languages were inherited from conventions that had evolved in math.
- Arithmetic expressions consist of operators, operands, parentheses, and function calls.

- The operators can be unary, or binary. C-based languages include a ternary operator, which has three operands (conditional expression).
- fThe purpose of an arithmetic expression is to specify an arithmetic computation.
- fAn implementation of such a computation must cause two actions:
  - o Fetching the operands from memory
  - o Executing the arithmetic operations on those operands.

Design issues for arithmetic expressions:

1. What are the operator precedence rules?
2. What are the operator associativity rules?
3. What is the order of operand evaluation?
4. Are there restrictions on operand evaluation side effects?
5. Does the language allow user-defined operator overloading?
6. What mode mixing is allowed in expressions?

Operator Evaluation Order

1. Precedence

- *f*The operator precedence rules for expression evaluation define the order in which "adjacent" operators of different precedence levels are evaluated ("adjacent" means they are separated by at most one operand).
- *f*Typical precedence levels:
  1. parentheses
  2. unary operators
  3. ** (if the language supports it)
  4. *, /
  5. +, -

- Many languages also include unary versions of addition and subtraction.
- *f* Unary addition (+) is called the identity operator because it usually has no associated operation and thus has no effect on its operand.
- *f* In Java, unary plus actually does have an effect when its operand is short or byte. An implicit conversion of short and byte operands to int type takes place.
- *f* Unary minus operator (-) Ex:

A + (- B) * C // is legal

A + - B * C // is illegal

2. Associativity

- The operator associativity rules for expression evaluation define the order in which adjacent operators with the **same precedence** level are evaluated. An operator can be either left or right associative.
- Typical associativity rules:
  - o Left to right, except **, which is right to left
  - o Sometimes unary operators associate right to left (e.g., FORTRAN)
- Ex: (Java)

```
a – b + c // left to right
```

- Ex: (Fortran)

  A ** B ** C // right to left

  (A ** B) ** C // In Ada it must be parenthesized

Language	Associativity Rule
FORTRAN	Left: * / + -
	Right: **
C-BASED LANGUAGES	Left: * / % binary + binary -
	Right: ++ -- unary – unary +
ADA	Left: all except **
	Non-associative: **

- **APL** is different; **all** operators have **equal** precedence and all operators associate **right to left**.
- Ex:

```
A X B + C // A = 3, B = 4, C = 5
27
```

- Precedence and associativity rules can be overridden with parentheses.

3. Parentheses

- Programmers can alter the precedence and associativity rules by placing parentheses in expressions.
- A parenthesized part of an expression has precedence over its adjacent un-parenthesized parts.
- Ex:

(A + B) * C

4. Conditional Expressions

- Sometimes **if-then-else** statements are used to perform a conditional expression assignment.

```
if (count == 0)
        average = 0;
else
        average = sum / count;
```

- In the C-based languages, this can be specified more conveniently in an assignment statement using a conditional expressions. Note that ? is used in conditional expression as a ternary operator (3 operands).

```
expression_1 ? expression_2 : expression_3
```

- Ex:

```
average = (count == 0) ? 0 : sum / count;
```

Operand evaluation order

- The process:

  1. Variables: just fetch the value from memory.

  2. Constants: sometimes a fetch from memory; sometimes the constant is in the machine language instruction.

  3. Parenthesized expressions: evaluate all operands and operators first.

- **Side Effects**

A **side effect** of a function, called a **functional side effect**, occurs when the function changes either one of its parameters or a global variable.

Ex:

a + fun(a)

- If fun does not have the side effect of changing a, then the order of evaluation of the two operands, a and fun(a), has no effect on the value of the expression.
- However, if fun changes a, there is an effect.

- Ex:

Consider the following situation: fun returns the value of its argument divided by 2 and changes its parameter to have the value 20, and:

```
a = 10;
b = a + fun(a);
```

- If the value of a is returned first (in the expression evaluation process), its value is 10 and the value of the expression is 15.
- But if the second is evaluated first, then the value of the first operand is 20 and the value of the expression is 25.
- The following shows a **C** program which illustrate the same problem.

```
int a = 5;
int fun1() {
a = 17;
return 3;
}
void fun2() {
a = a + fun1(); // C language a = 20; Java a = 8
}
void main() {
fun2();
}
```

- The value computed for a in fun2 depends on the order of evaluation of the operands in the expression a + fun1(). The value of a will be either 8 or 20.
- Two possible solutions:

1. Write the language definition to disallow functional side effects

   o No two-way parameters in functions.

   o No non-local references in functions.

   o **Advantage**: it works!

   o **Disadvantage**: Programmers want the flexibility of two-way parameters (what about C?) and non-local references.

2. Write the language definition to demand that operand evaluation order be fixed

   o **Disadvantage**: limits some compiler optimizations

Java guarantees that operands are evaluated in **left-to-right order**, eliminating this problem. **// C language a = 20; Java a = 8**

Overloaded Operators

- The use of an operator for **more than one purpose** is operator overloading.
- Some are common (e.g., + for int and float).
- Java uses + for addition and for **string catenation**.
- Some are potential trouble (e.g., **&** in C and C++)

```
x = &y // as binary operator bitwise logical

// AND, as unary it is the address of y
```

– Causes the address of y to be placed in x.

– Some loss of readability to use the same symbol for two completely unrelated operations.

– The simple keying error of leaving out the first operand for a bitwise AND operation can go undetected by the compiler “difficult to diagnose”.

– Can be avoided by introduction of new symbols (e.g., Pascal’s **div for integer division and / for floating point division**)

Type Conversions

• A **narrowing conversion** is one that converts an object to a type that cannot include all of the values of the original type e.g., **double to float**.

• A **widening conversion** is one in which an object is converted to a type that can include at least approximations to all of the values of the original type e.g., **int** to **float**.

Coercion in Expressions

- A **mixed-mode expression** is one that has operands of different types.
- A **coercion** is an implicit type conversion.
- The disadvantage of coercions:
- They decrease in the type error detection ability of the compiler
- In most languages, all numeric types are coerced in expressions, using widening conversions
- Language are not in agreement on the issue of coercions in arithmetic expressions.
- Those against a broad range of coercions are concerned with the reliability problems that can result from such coercions, because they eliminate the benefits of type checking.

• Those who would rather include a wide range of coercions are more concerned with the loss in flexibility that results from restrictions.

• The issue is whether programmers should be concerned with this category of errors or whether the compiler should detect them.

• Java method Ex:

```
void mymethod() {
int a, b, c;
float d;
…
a = b * d;
…
}
```

- Assume that the second operand was supposed to be c instead of d.
- Because mixed-mode expressions are legal in Java, the compiler would not detect this as an error. Simply, b will be coerced to **float**.

Explicit Type Conversions

- Often called casts in C-based languages.
- Ex: Ada:

FLOAT(INDEX)--INDEX is INTEGER type

Java:

(int)speed /*speed is float type*/

Errors in Expressions

- Caused by:
  - Inherent limitations of arithmetic e.g. division by zero
  - Limitations of computer arithmetic e.g. overflow or underflow
- Floating-point overflow and underflow, and division by zero are examples of **run-time errors**, which are sometimes called exceptions.

Relational and Boolean Expressions

• A relational operator: an operator that compares the values of its tow operands.

• Relational Expressions: two operands and one relational operator.

• The value of a relational expression is **Boolean**, unless it is not a type included in the language.

– Use relational operators and operands of various types.

– Operator symbols used vary somewhat among languages (!=, /=, .NE., <>, #)

• The syntax of the relational operators available in some common languages is as follows:

Operation	Ada	C-Based Languages	Fortran 95
Equal	=	==	.EQ. or ==
Not Equal	/=	!=	.NE. or <>
Greater than	>	>	.GT. or >
Less than	<	<	.LT. or <
Greater than or equal	>=	>=	.GE. or >=
Less than or equal	<=	<=	.LE. or >=

Boolean Expressions

• Operands are Boolean and the result is **Boolean**.

FORTRAN 77	FORTRAN 90	C	Ada
.AND.	and	&&	and
.OR.	or	||	or
.NOT.	not	!	not

• Versions of **C** prior to C99 have no Boolean type; it uses int type with **0 for false and nonzero for true**.

• One odd characteristic of C's expressions:

a < b < c is a legal expression, but the result is not what you might expect.

• The left most operator is evaluated first because the relational operators of C, are left associative, producing **either 0 or 1**.

• Then this result is compared with var c. There is **never** a comparison between b and c.

Assignment Statements

Simple Assignments

- The C-based languages use == as the equality relational operator to avoid confusion with their assignment operator.
- The operator symbol for assignment:
  1. = FORTRAN, BASIC, PL/I, C, C++, Java
  2. := ALGOL, Pascal, Ada

Conditional Targets

- Ex:

flag ? count 1 : count2 = 0; ⇔ **if** (flag)

count1 = 0;

else

count2 = 0;

Compound Assignment Operators

- A compound assignment operator is a shorthand method of specifying a commonly needed form of assignment.
- The form of assignment that can be abbreviated with this technique has the destination var also appearing as the first operand in the expression on the right side, as in

$$a = a + b$$

- The syntax of assignment operators that is the catenation of the desired binary operator to the = operator.

$$\text{sum += value;} \Leftrightarrow \text{sum = sum + value;}$$

Unary Assignment Operators

- C-based languages include two special unary operators that are actually abbreviated assignments.
- They combine increment and decrement operations with assignments.
- The operators ++ and -- can be used either in expression or to form standalone single-operator assignment statements. They can appear as prefix operators:

sum = ++ count; ⇔ count = count + 1; sum = count;

- If the same operator is used as a postfix operator:

sum = count ++; ⇔ sum = count; count = count + 1;

Assignment as an Expression

- This design treats the assignment operator much like any other binary operator, except that it has the side effect of changing its left operand.
- Ex:

```
while ((ch = getchar())!=EOF)
{…} // why ( ) around assignment?
```

- The assignment statement must be parenthesized because the precedence of the assignment operator is lower than that of the relational operators.

- Disadvantage: Another kind of expression side effect which leads to expressions that are difficult to read and understand. For example

a = b + (c = d / b++) – 1

denotes the instructions

Assign b to temp

Assign b + 1 to b

Assign d / temp to c

Assign b + c to temp

Assign temp – 1 to a

- There is a loss of error detection in the C design of the assignment operation

that frequently leads to program errors.

```
if (x = y) …
```

instead of

```
if (x == y) …
```

Mixed-Mode Assignment

- In FORTRAN, C, and C++, any numeric value can be assigned to any numeric scalar variable; whatever conversion is necessary is done.
- In Pascal, integers can be assigned to reals, but reals cannot be assigned to integers (the programmer must specify whether the conversion from real to integer is truncated or rounded.)
- In **Java**, only **widening** assignment coercions are done.
- In **Ada**, there is no assignment coercion.
- In all languages that allow mixed-mode assignment, the coercion takes place **only after** the right side expression has been evaluated. For example, consider the following code:

```
int a, b;
float c;
…
c = a / b;
```

Because c is float, the values of a and b could be coerced to float before the division, which could produce a different value for c than if the coercion were delayed (for example, if a were 2 and b were 3).