PROGRAMMING PARADIGM

MODULE -2

Data Types – Primitive Data Types, Character String Types, User-Defined Ordinal Types, Array Types, Record Types, List Types, Pointer & Reference Types, Type Checking, Strong Typing, Type Equivalence. Expressions – Arithmetic Expressions, Overloaded Operators, Type Conversions, Relational and Boolean Expressions, Short-Circuit Evaluation. Assignment - Assignment Statements, Mixed-mode assignment

1. 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

2. 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

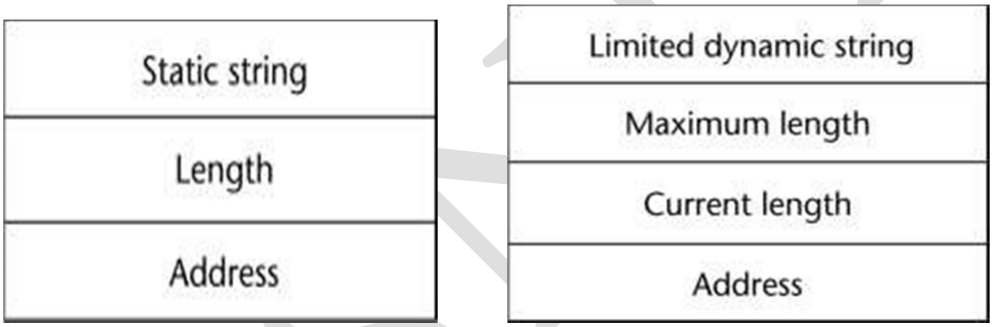
- Aid to writability

- 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

3. 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

4. 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
- *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)
- 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)

- 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 lements

– 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

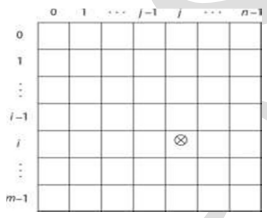
  mat[0][0:2] is the first and second element of the first row of mat

- Ruby supports slices with the slice method list.slice(2, 2) returns the third and fourth elements of list

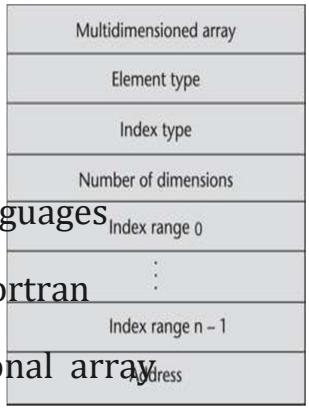
Implementation of Arrays

- 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

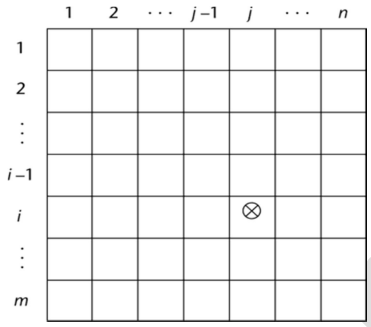


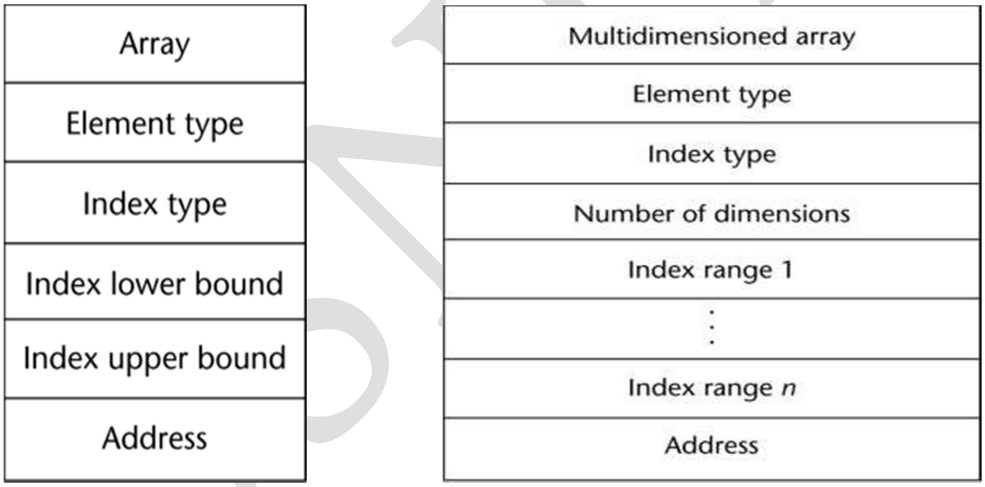
- Two common ways:
  - Row major order (by rows) – used in most languages
  - Column major order (by columns) – used in Fortran
  - A compile-time descriptor for a multidimensional array

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





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 **%; l**iterals 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"};

5. 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

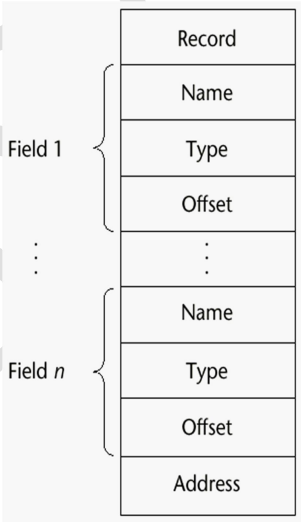
- Assignment is very common if the types are identical
- Ada allows record comparison

- 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**

- 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)

6. 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 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 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
- 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"]

- 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]

- 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*

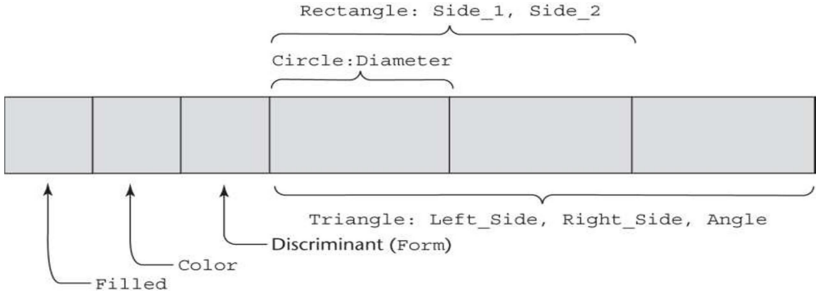
– Supported by Ada

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;

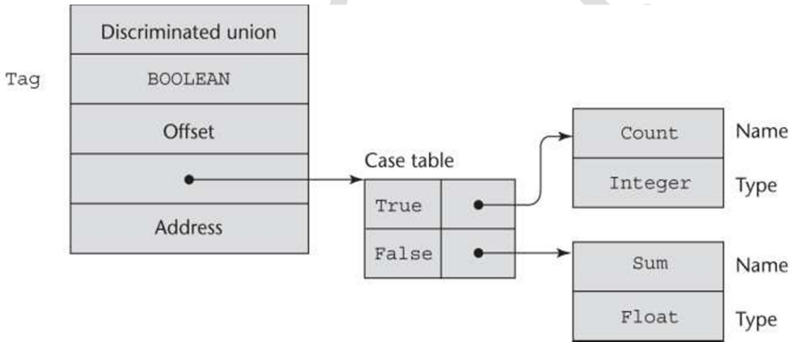
Ada Union Type Illustrated



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

7. 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?

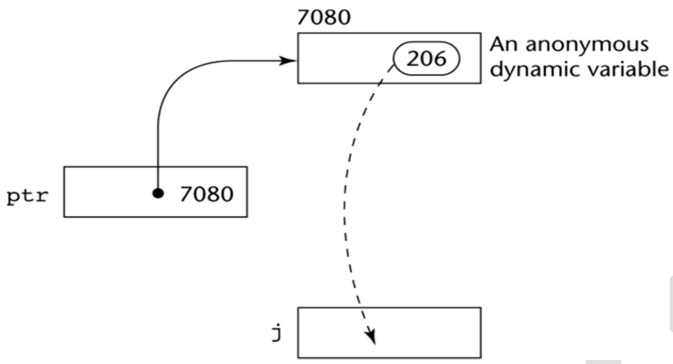
Pointer Operations

- 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 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

A reference type variable is similar to a pointer, with one important and fundamental difference: A pointer refers to an address in memory, while a reference refers to an object or a value in memory

- 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

Heap Management

- A very complex run-time process
- Single-size cells vs. variable-size cells
- Two approaches to reclaim garbage
  - Reference counters (*eager approach*): reclamation is gradual
  - Mark-sweep (*lazy approach*): reclamation occurs when the list of variable space becomes empty

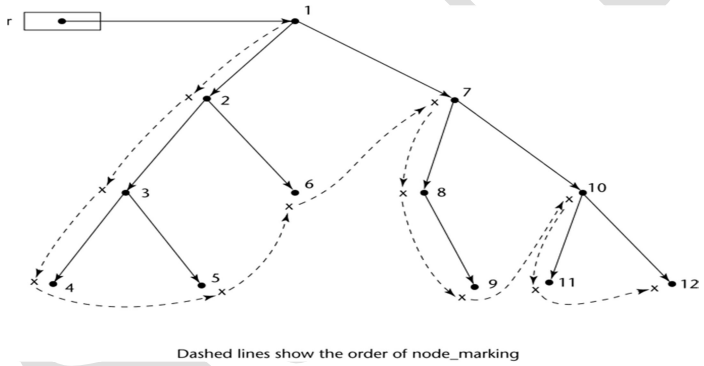
Reference Counter

- Reference counters: maintain a counter in every cell that store the number of pointers currently pointing at the cell
  - *Disadvantages*: space required, execution time required, complications for cells connected circularly
  - *Advantage*: it is intrinsically incremental, so significant delays in the application execution are avoided

Mark-Sweep

- The run-time system allocates storage cells as requested and disconnects pointers from cells as necessary; mark-sweep then begins
  - Every heap cell has an extra bit used by collection algorithm
  - All cells initially set to garbage
  - All pointers traced into heap, and reachable cells marked as not garbage
  - All garbage cells returned to list of available cells
  - Disadvantages: in its original form, it was done too infrequently. When done, it caused significant delays in application execution. Contemporary mark-sweep algorithms avoid this by doing it more often—called incremental mark-sweep

Marking Algorithm



Dashed lines show the order of node_marking

Variable-Size Cells

- All the difficulties of single-size cells plus more
- Required by most programming languages
- If mark-sweep is used, additional problems occur
  - The initial setting of the indicators of all cells in the heap is difficult
  - The marking process in nontrivial

– Maintaining the list of available space is another source of overhead

8. 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
- 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

9. Strong Typing

A programming language is strongly typed if type errors are always detected. A strongly typed language also allows the detection, at run time, of uses of the incorrect type values in variables that can store values of more than one type.

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)
- 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

10. Type Equivalence

Type equivalence is a strict form of type compatibility—compatibility without coercion. There are two approaches to defining type equivalence: name type equivalence and structure type equivalence.

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
- 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
- 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

Summary

- The data types of a language are a large part of what determines that language's style and usefulness

- The primitive data types of most imperative languages include numeric, character, and Boolean types
- The user-defined enumeration and subrange types are convenient and add to the readability and reliability of programs
- Arrays and records are included in most languages
- Pointers are used for addressing flexibility and to control dynamic storage management

11. Expressions

Expressions are the fundamental means of specifying computations in a programming language. It is crucial for a programmer to understand both the syntax and semantics of expressions of the language being used.

Arithmetic expressions

- Consists of operators, operands, parenthesis and function calls
- Purpose is to specify an arithmetic computation
- An operator can be

  Unary: single operand

  Binary: two operands

  Tertiary: three operands
- In most programming languages. Binary operators are **infix** which means they appear between their operators
- The exception to this is Perl which has some operators that are **prefix**
- Two actions for computation: fetching operands from memory and executing arithmetic operations on those operands

  Operation evaluation order

  Precedence
- a + b * c

- suppose the values are 3,4,5 respectively
- If evaluated left to right: 35
- If evaluated right to left: 23
- Instead of this, use operator precedence rules which define the order in which the operators of different precedence levels are evaluated
- Precedence rules for common imperative languages are nearly al the same because it is based on mathematics where exponentiation has the highest precedence, followed by mul and div on the same level followed by binary addition and subtraction on the same level
- Many languages also include unary versions of addition and subtraction
- Unary addition is called identity operator because it usually has no associated operation and thus has no effect on its operand
- Unary minus changes sign of its operand
- In Java and C# unary minus also causes the implicit conversion of **short** and **byte** operands to **int** type
- A + (-b) * c is legal but a+-b*c is not legal
- -a/b -a*b -a**b
- First two cases, relative precedence of unary minus and binary operator is irrelevant however in the last cause it does matter as it has an effect on the value of the operation
- Only Fortran, Ruby, Visual Basic and Ada have exponentiation operator. In all 4, it has a higher precedence than unary minus
- -a**b is equivalent to –(a**b)
- Precedence's for ruby and C-based languages are as follows (from highest to lowest)
- Ruby: **, (unary +, -), (*, /, %), (binary +, -)

- C-based: (postfix ++,--), prefix(++x--), (*, /, %), (binary +, -)

Associativity

- A-b+c-d
- Two operators are adjacent if they are separated by a single operand
- When an expression contains two adjacent occurrences of operators with the same level of precedence, the question of which must be evaluated first is answered by the associativity rules
- Left or right associativity, ie if two adjacent operators, left operator is evaluated first or right operator is evaluated first
- Common languages is left to right except that the exponentiation operator sometimes
- associates right to left
- Fortran and Ruby
- Exponentiation is right associative
- A**B**C
- Right operator is evaluated first
- Visual basic
- Exponentiation operator, ^ , is left associative.

Language	*Associativity Rule*
Ruby	Left: *, /, +, –
	Right: **
C-based languages	Left: *, /, %, binary +, binary –
	Right: ++, --, unary –, unary +

• APL

o All operators have the same level of precedence thus order of evaluation of operators are determined entirely by associativity rule.

o It is left to right for all operators

o Eg) A*B+C

o If A=3, B=4 and C=5 then it will be **27**

• Many compilers make use of the fact that some arithmetic operators are mathematically associative meaning that the associativity rules have no impact on the value of an expression containing only those operators

• For example, addition is mathematically associative so the value of the expression **A+B+C** does not depend on the order of operation evaluation

• IF floating point operations for mathematically associative operations were also associative, the compiler could use this fact to perform simple optimizations. Specifically, if operations could be reordered by compiler, it may be able to produce slightly faster code for expression evaluation

• Compilers commonly do these kinds of operations.

• Unfortunately, floating point representations and floating-point operations or only approximations due to size limitations

The fact that mathematical operator is associative DOES NOT mean corresponding floating-point operator is associative

ONLY if all operands and intermediate results can be exactly represented in floating point representation will the process be precisely associative.

Pathological situations in which integer addition is not associative

Eg A+B+C+D

- Assume A and C are very large positive numbers and B and D are negative numbers with very large absolute values
- Adding B to A does not cause overflow but C to A does.
- Because of limitations, addition is non-associative in this case.

▪ Problem can be avoided by programmer by specifying the expression in two parts ensuring that overflow is avoided

▪ However, may occur in more subtle ways where programmer is less likely to Notice Parentheses

• Precedence and association can be altered by placing parentheses

• Parenthesized part has precedence over its adjacent unparenthesized parts Eg (A+B)*C

• Addition will be done first

• Also (A+B) + (C+D) could be done to avoid overflow

• Languages that allow parentheses in arithmetic operations could dispose of all precedence rules and simply associate all operators left to right or right to left. Seems simple because no precedence rules or associativity will need to be remembered

• Disadvantage of this scheme is that it makes writing expressions more tedious and compromises readability of code

•Yet this was the choice made by Ken, the designer of **APL**

Ruby expressions

• Ruby is a pure OO language which means among other things, every data value, including literals is an object

• It supports collection of arithmetic and logic included in C-based languages

• What sets it apart is that all arithmetic, relational and assignment operators as well as array indexing, shifts and bitwise logic operators are implemented as **methods**

Eg a+b is a call to the + method of the object referenced by a passing the object referenced by b as a parameter

• Implementation of operators as methods means that they can be **overridden** by application programs. Therefore these operations can be redefined

Lisp expressions

• All arithmetic and logic operators performed by subprograms. Subprograms must be explicitly called.

Eg a+b*c

(+a(*bc))

+ and * are the names of the functions

Conditional expressions

• If-then-else

• **if** (count == 0)

average = 0;

else

average = sum / count;

• In C based languages it can be specified more conveniently as

o **Expression1 ? Expression2 : expression3**

• Where expression1 is interpreted as a Boolean expression

o If true, value of expression2

o Else value of expression3

o **Average = (count==0) ? 0 :sum/count;**

• Also provided in Perl, JavaScript and Ruby

Operand Evaluation order

• Variables in expressions are evaluated by fetching their values from memory

• Constants are sometimes evaluated in the same way. In other cases, a constant may be part of the machine language instruction and not require a memory fetch

• If operands of an operator has side effects then operand evaluation order is irrelevant

Side effects

• Occurs when a function changes either one of its parameters or a global variable

• A + fun(a)

• If fun does not have the side effect of changing a then order of evaluation has no effect on the value

• If fun changes a then there is an effect

• Fun returns 10 and changes the value of its parameter to 20

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

o Then if a is fetched first, its value is 10 and expression is 20

o If second operand is evaluated first then first operand value is 20 and the value of the expression is 30

• C-program illustrates same problem when a function changes a global variable that appears in an expression

```
int a = 5;
int fun1() {
o
 a = 17;
 return  3;
}   /* end of fun1 */
void  main() {
 a = a + fun1();
}   /* end of main */
```

• The value of a will wither be 8(if a is evaluate first) ir 20 (if function call is evaluated first)

• Two solutions

o Language designer could disallow function evaluation from affecting the value of expressions by simply disallowing functional side effects

o Language definition could state that operands in expressions are to be evaluated in a particular order and demand that implementers guarantee that order

• Disallowing functional side effects in imperative languages is difficult and it eliminates some flexibility for the programmer

• Consider C and C++, which have only functions meaning all subprograms return one value

o To eliminate the side effects of two-way parameters and still provide subprograms that return more than one value, the values would need to be placed in a struct and the struct returned.

o Access to global in functions would also have to be disallowed

o However when the efficiency is important, using access to global variables to avoid

parameter passing is an important method of increasing execution speed

• The problem with having strict evaluation order is that some code optimization techniques used by compilers involve reordering operand evaluations

• A guaranteed order disallows those optimization methods when function calls are involved

• No perfect solution

• **Java** guarantees that operands appear to be evaluated in left to right order, eliminating the problem discussed in this section Referential transparency and side effects

• A program has the property of referential transparency id any two expressions in the program that have the same value can be substituted for one another anywhere in the program without affecting the action od the program

```
result1 = (fun(a) + b) / (fun(a) - c);
temp = fun(a);
result2 = (temp + b) / (temp - c);
```

• If the function fun has no side effects then result1 and result2 will be equal

• Supposed fun has the side effect of adding 1 to either b or c. result1 would not be equal to result2 so that side effect violates the referential transparency of the program

• Advantages

• Semantics of such programs is much easier to understand than the semantics of programs that are not referentially transparent. Being referentially transparent makes a function equivalent to a mathematical function in terms of ease of understanding o Programs written in pure functional languages are referentially transparent because they do not have variables. Functions in pure functional language cannot have state which would be stored in local variables. If such a function uses a value from outside, value must be a constant since there are no variables. Therefore the value of the function depends on the values of its parameters.

12. Overloaded operators

• + used for integer addition and floating-point addition. Some languages also use it for string catenation (**Java**)

• Multiple use of a operator is called operator overloading and is generally thought to be acceptable

• Eg of possible dangers of operator overloading:

Consider the use of &

- As a binary operator, specifies a bitwise logical AND operation
- As a unary, it is the address of operator

• (x= &y)

• Causes address of y to be placed in x

• Two problems

- Detrimental to readability since same sign used for two completely different unrelated operations
- Simple keying error of leaving out the first operand for a bitwise AND operation can go undetected by the compiler because it is interpreted as an address of operator

• Some languages that support abstract data types, eg C++, C# and F# allow futher overload operator symbols

• Eg) * can be defined for a scalar integer and integer array to mean that each element of the array is to be multiplied by the scalar

•Could be done by writing a function subprogram named * that performs this new operation

• Compiler will choose the correct meaning when overloaded operator is specified based on the types of the operands

• Can aid readability if used sensibly

• Eg) A*B+C*D can be used instead of MatrixAdd(MatricMult(A,B), MatrixMult(C,D))

• Could also be harmful to readability

▪ For one thing, nothing prevents user from defining + to mean multiplication

▪ Furthermore seeing a * operator in a program, the reader must fine both types of the operands and the definition of the operator to determine its meaning. Any or all of these definitions could be in other files

• Another consideration is when building a software system with different groups. If the different groups overloaded the same operators in different ways, these differences would need to be eliminated before putting the system together

• C++ has a few operators that cannot be overloaded such as the class or structure member operator (.) and scope resolution operator (::)

• Operator overloading was not copied from C++ to java but it did reappear in C#

13. Type conversions

• Either narrowing or widening

• Narrowing conversion: Converts a value to a type that cannot store even approximations of all the values of the original type.

o Eg) double to float in Java is narrowing because the range of double is much larger than float

• Widening conversion: converts a value to a type that can include at least approximations of all the values of the original type.

• Int to float in Java is widening

• Widening conversions are nearly always safe meaning that the approximate magnitude of the converted value is maintained

• Narrowing are not always safe since magnitude of the converted value is changed in the process.

Eg. floating point value 1.3E25 converted to int in Java will not be in any way related to the original value

• Although widening is usually safe, it can result in reduced accuracy.

Eg. int stored in 32 bits which allows at least 9 decimal digits but floating point is also stored in 32 bits with only 7 decimal digits of precision (space used for exponent)

•So, it could result in loss of two digits of precision

• Coercions of non-primitive types are more complex

• Also, the question of what parameter types and return types of a method allow it to override a method in a superclass. Only wen types are the same or some other situations

• Type conversions are either implicit or explicit

Coercion in expressions

• One of the design decisions concerning arithmetic expressions is whether an operator can have operands of different types. Languages that allow such expressions, which are called

mixed-mode expressions, must define conventions for implicit operand type conversions because computers do not have binary operands of different types

• Coercion is an implicit type conversion that is initiated by the compiler or runtime system

• Type conversions explicitly requested by the programmer are referred to **as explicit conversions or casts,** not coercions

• For overloaded operators in a language that uses static type binding, the compiler chooses the correct type of operation on the basis of the types of the operands. When two operands are not the same type, compiler must choose one to be coerced and generate the code for that coercion

• Language designers not in agreement on issue of coercion

•Some are against a broad range of coercion as they are concerned with reliability problems since reduce benefits of type checking

• Some would rather include broad range as they are more concerned with loss in flexibility that results from restrictions

• Issue is whether programmers need to be concerned with errors or compiler should detect them

• Eg int a;

float b,c,d;

…….

d-b*a;

• Assume second operand of multiplication was supposed to be c by mistakenly typed as a

• Because mixed-mode expressions are legal in **Java,** compiler would not see it as an error, it would simply insert code to coerce value of int operand a to float

• **F#** and **ML** does not allow mixed mode as error detection is reduced

• C-based languages such as Java have short and byte which are smaller than int. these are coerced into int.

• Eg byte a,b,c;

• A=b+c;

• B and c coerced into int, int addition performed then sum converted to byte and stored in a

Explicit type conversions

• Most languages provide some capability for doing explicit conversions, both widening and narrowing

• In most cases, warning message produced when narrowing conversion results in significant change to result

• In C-based languages, explicit type conversions are known as **casts**

• (int) angle; //cast int is desired type

• Reason for parentheses is because C has several two-word type names such as **long int**

• ML and F# the casts have the syntax of function calls

• Float(sum)

Errors in expressions

• A number of errors can occur during expression evaluation

• If language requires type checking, either static or dynamic then operand type errors cannot occur

• Other type of errors is due to limitations of computer arithmetic and the inherent limitations of arithmetic

• Most common error is when result of an operation cannot be represented in memory cell where it must be stored. Called **overflow** or **underflow**, depending on whether result is too large or too small

• One limitation is that division by zero is disallowed

• Floating point overflow, underflow and division by zero are examples of run-time errors which are sometimes called **exceptions**

14. Relational expressions

• **Relational operator** is an operator that compares the values of two operands

• Value is Boolean except when Boolean is not a type included in the language

• It is often overloaded for a variety of types

• It can be simple as for integer operands or complex as for character string operands

• Syntax for equality and inequality differs among some programming languages. For inequality:

C- based uses !=

Lua used ~=

Fortran 95+ uses .NE. or <>

ML and F# uses <>

• JavaScript and PHP have two additional relational operators, === and !== which are similar to == and != but prevent operands from being coerced

o Eg) "7"==7 is true in javascript because string coerced into a number

• "7" === 7 is false because no coercion is done

• Ruby uses == for equality that includes coercion and eql? For equality with no coercions

• Relational operators always have **LOWER** precedence than arithmetic operators so

• A+1 > 2*b

• Arithmetic operations evaluated first

15. Boolean expressions

• Consists of Boolean variables, Boolean constants, relational expressions and Boolean operators

• AND, NOT, OR and sometimes XOR and equivalence

• Boolean operators usually take on Boolean operands and produce Boolean values

• In mathematics of Boolean algebras, OR and AND must have equal precedence however n Cbased
languages AND has a higher precedence than OR

Highest	postfix ++, --
	unary +, unary -, prefix ++, --, !|
	*, /, %
	binary +, binary -
	<, >, <=, >=
	=, !=
	&&
Lowest	||

• Versions of C prior to C99 are odd among popular imperative languages they have no Boolean type and thus no Boolean values. Instead numeric values are used with zero considered false and all nonzero values true

• The odd result of C's design is that the following is legal

• A>b>c

• Leftmost operator evaluated first producing either 0 or 1 then that result is compared with c. there is never a comparison between b and c

• Some languages including Perl and Ruby, provide two sets **&&** and **and** for AND and **||** and **or** for OR.

• The difference is that the spelled versions have lower precedence.

• Also, **and** and **or** have equal precedence but **&&** has higher precedence than **||**

16. Short circuit evaluation

• Short circuit evaluation of an expression is one in which the result is determined without evaluating all of the operands and/or operators.

• Eg) (13 * a) * (b / 13 - 1)

▪ Is independent of the value of (b / 13 - 1) if a is 0 because 0*x=0 for any x

▪ So when a is 0 there is no need to calculate (b / 13 - 1) or perform the second multiplication. However, this is not easily detected so it is never taken

• Boolean expression (a >= 0) && (b < 10)

▪ Is independent of second relational expression if a<0

▪ This can be easily detected

• Potential problem with non-short circuit evaluation. Suppose java did not use it

• A table lookup loop could be written using the while statement

```
index =0;
while ((index < listlen)  && (list[index] !-key))
index = index + 1;
```

• Assuming list has listlen elements is the array to be searched for and key is the searched for value

• If evaluation is not short circuit, both relational expressions in the Boolean expression of the while statement are evaluated regardless of the value of the first

• Thus if key is not in list, program will terminate with the out-of-range expression

• A language that provides short-circuit evaluations of Boolean expressions and also has side effects in expressions allows subtle errors to occur

•(a>b) || (b++) / 3)

•In this expression, b is changed only when a<=b. if the programmer assumer b will be changed every time this expression is evaluated during execution then the program will fail

• In the C-based languages, the usual AND and OR operators, && and ||, respectively, are short-circuit. However, these languages also have bitwise AND and OR operators, & and |, respectively, that can be used on Boolean-valued operands and are not short-circuit. Of course, the bitwise operators are only equivalent to the usual Boolean operators if all operands are restricted to being either 0 (for false) or 1 (for true).

• All of the logical operators of Ruby, Perl, ML, F#, and Python are short-circuit evaluated.

17. **Assignment : Assignment statements**

• Assignment statement provides the mechanism by which the user can dynamically change the bindings of values to variables

Simple assignments

• Nearly all programming languages currently being used use the equal sign for assignment operator

• All of these must use something different from and equal sign for the equality relational operator to avoid confusion with their assignment operator

• ANGOL60 pioneered the use of := as assignment operator

• Design choices of how assignments are used in a language have varied widely. Some languages such as Fortran and Ada an assignment can appear

only as a stand-alone statement and the destination restricted to a single variable

Conditional targets

• Perl allows conditional targets on assignment statements eg)

```
($flag ? $count1 : $count2) =0;
```

Which is equivalent to

```
If($flag){
$count1-0;
}else  {
$count2 =0;
}
```

Compound assignment operators

• Shorthand method of specifying a commonly needed form of assignment

• The form: a=a+b where destination variable also appearing as first operand

• Introduced by ALGOT68 were later adopted in a slightly different form by C and are part of

other C-based languages as well as Perl, JavaScript, Python and Ruby.

• Syntax of these assignment operators is the catenation of the desired binary operator to the

= operator

• Sum+= value;

Unary assignment operators

• C-based languages, Perl and JavaScript include two unary arithmetic operators that are actually abbreviated assignments. Combine increment and decrement operators with assignment

• ++ and –

• Can appear either as prefix (precede operands) or postfix (follow operands) operators

• Sum=++count // count incremented by 1 and assigned to sum

• Same as count=count+1; sum=count;

• Sum = count++; //assignment of count to sum occurs first then count is incremented

• Same as sum=count; count=count+1;

• When two unary operators apply to the same operand, the association is **right to left**

-count ++

• Count is first incremented then negated so it is the same as –(count++) rather than (-count)++

Assignment as an expression

• C-based languages, Perl, and JavaScript, assignment statement produces a result which is the same as the value assigned to the target

• It can therefore be used as an expression and as an operand in other expressions

While((ch=getchar()!=EOF) {…}

• In this statement, next char from the standard input file, usually the keyboard is gotten with getchar and assigned to ch

• If ch is not equal to EOF the compound statement {..} is executed

• NOTE!! Assignment must be parenthesized- in the languages that support assignment as an expression, the precedence of the assignment operator is lower than that of the relational operators. Without parenthesis, new char would be compared with EOF first then result, either 0 or 1 would be assigned to ch

• Disadvantage of allowing this expresses yet another kind of side effect

Leads to an expression that is difficult to read and understand. An expression with any kind of side effect has this disadvantage

Such an expression cannot be read as an expression which in mathematics is a denotation of a value but only as a list of instructions with an odd order of execution

Eg

```
a = b + (c = d / b) - 1
```

denotes the instructions

```
Assign d / b to c
Assign b + c to temp
Assign temp - 1 to a
```

• Treatment of assignment operator as any other binary operator allows effect of multiple target assignments

• Sum=count=0 //0 assigned to count, count assigned to sum

• Legal in python

• Loss of error detection in C design of assignment operation that frequently leads to program errors.

• If we type if(x=y) instead of if(x==y)

•It is not detectible as an error by compiler

• This is the result of two design decisions

- Allowing assignments to behave like an ordinary binary operator
- Using two very similar operators = and == to have completely different meanings
- This is another example of safety deficiency of C and C++ programs

• Java and C# allow only Boolean expressions in their if statements, disallowing this problem

Multiple assignments

• Perl, Lua and Ruby provide multiple target, multiple source assignment statements.

Eg in Perl one can write

- ($first, $second, $third) = (20,40,60);
  If two variables need to be interchanged, it can be done by
- ($first, $second) = ($second, $first)
  This is done without use of temporary variable

• Ruby's multiple assignment same as Perl except left and right sides are not parenthesized

Assignment in functional programming languages

• All identifiers used in pure functional languages and some of them used in other functional languages are just names of values. As such their values never change

• Eg) ML names are bound to values with the **val** declaration

o **val** cost= quantity*price o if cost appears on the left side of val declaration that declaration creates a new version of the name cost which has np relationship with the prev version which is

then hidden

• F# has similar except uses keyword **let**

• Difference between F# let and ML val is that let creates a new scope whereas val does not

18. Mixed mode assignment

An expression that contains variables and constants of different data types is called as a mixed mode expression.

Assignment statements can also be mixed-mode, for example

Int a, b;

Float c;

C = a / b;

- **In Fortran, C, and C++, any numeric type value can be assigned to any numeric**

Type variable.

- **In Java, only widening assignment coercions are done.**

- **In Ada, there is no assignment coercion.**