Module 2

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

Datatypes

- A **data type** defines a collection of data values and a set of predefined operations on those values.
- Computer programs produce results by manipulating data.
- An important factor in determining the ease with which they can perform this task is how well the data types available in the language being used match the objects in the real-world of the problem being addressed.
- Therefore, it is crucial that a language supports an appropriate collection of data types and structures

- 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

1. Numeric Types

1.1 Integer

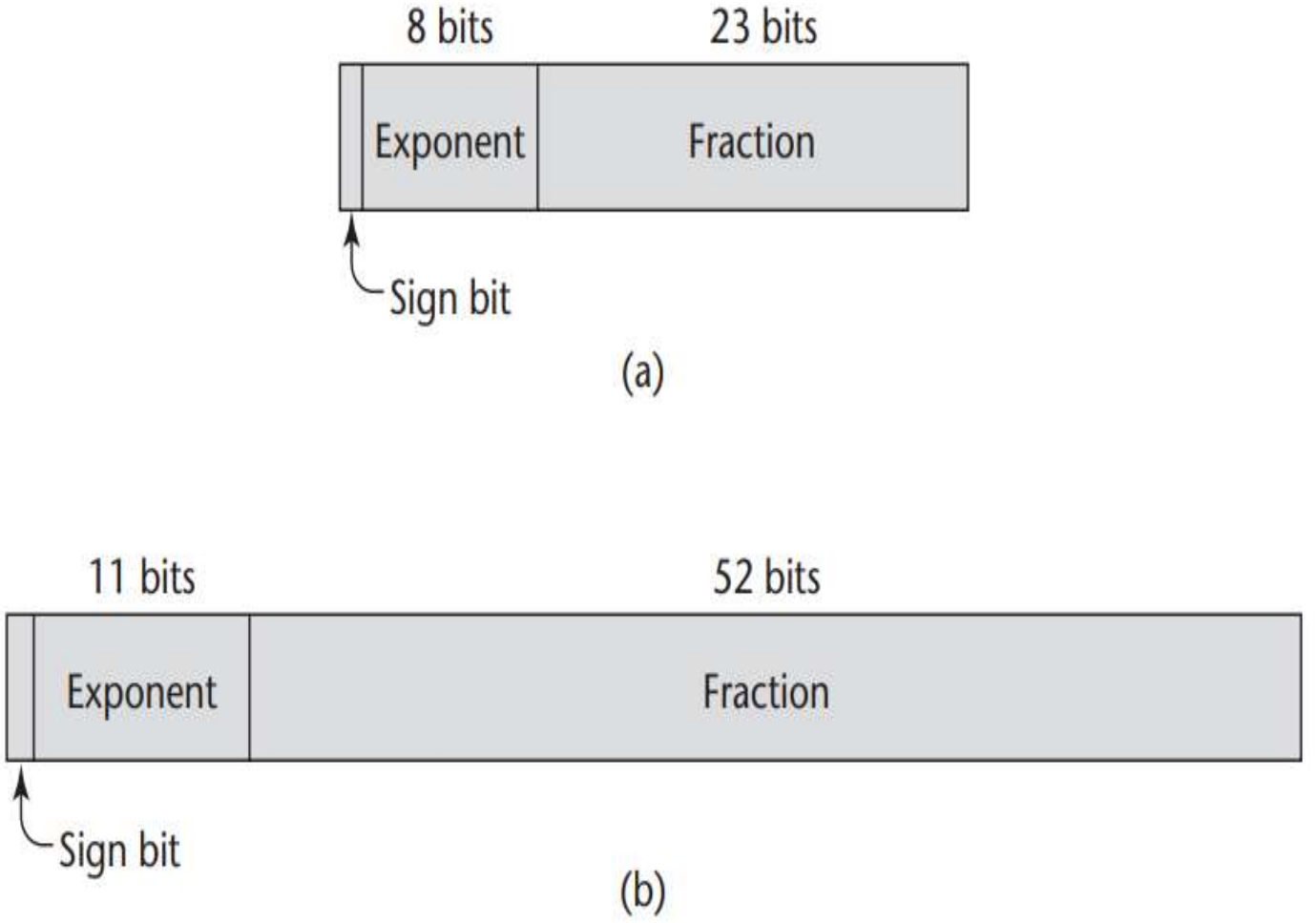
- Almost always an exact reflection of the hardware so the mapping is trivial
- There may be as many as eight different integer types in a language
- **Java‘s signed integer sizes: byte, short, int, long**
- A **signed integer value** is represented in a computer by a string of bits, with one of the bits (typically the leftmost) representing the sign.
- A negative integer could be stored in **sign-magnitude notation**. However, **twos complement** method etc are used to store negative integers, which is convenient for addition and subtraction

1.2 Floating - Point

- Most newer machines use the **IEEE Floating-Point Standard 754 format**.
- The **float type** is the standard size, usually being stored in **four bytes** of memory.
- The **double type** is provided for situations where larger fractional parts and/or a larger range of exponents is needed.
- Double-precision variables usually occupy twice as much storage as float variables and provide at least twice the number of bits of fraction
- The collection of values that can be represented by a floating-point type is defined in terms of **precision and range**.
- **Precision** is the accuracy of the fractional part of a value, measured as the number of bits.
- **Range** is a combination of the range of fractions and, more important, the range of exponents.

Figure 6.1

IEEE floating-point formats: (a) single precision, (b) double precision



1.3 Complex

- Some languages support a complex type, e.g., Fortran and Python
- Each value consists of two floats, the real part and the imaginary part.
- In Python, the imaginary part of a complex literal is specified by following it with a j or J.
- Eg: (7 + 3j), where 7 is the real part and 3 is the imaginary part

1.4 Decimal

- For business systems applications (money)
- Decimal data types store a fixed number of decimal digits, with the decimal point at a fixed position in the value.
- – Essential to COBOL
- – C# offers a decimal data type
- Decimal types are stored like character strings, using binary codes for the decimal digits. These representations are called **binary coded decimal (BCD).**
- They take more storage than binary representations.
- Advantage: accuracy
- Disadvantages: limited range, wastes memory

2. Boolean Types

- Simplest of all types
- Range of values has only two elements, one for **true** and one for **false**
- They were introduced in ALGOL 60 and have been included in most general-purpose languages designed since 1960.
- Boolean types are often used to represent switches or flags in programs.
- Could be implemented as bits, but often as bytes
- A Boolean value could be represented by a single bit, but they are often stored in a **byte**.
- Advantage: **readability**

3. Character Types

- Stored as numeric coding
- Most commonly used coding: ASCII
- An alternative, 16-bit coding: Unicode
- Includes characters from most natural languages
- – Originally used in Java
- – C# and JavaScript also support Unicode

Character String Types

- Values are sequences of characters.
- character strings are an essential type for all programs that do character manipulation
- **1. Design issues:**
- – Is it a primitive type or just a special kind of array?
- – Should strings have static or dynamic length?

2. Strings and Their Operations

- Typical operations:
- – Assignment and copying
- – Comparison (=, >, etc.)
- - Catenation
- – Substring reference – substring references are called **slices**.
- –Pattern matching is a fundamental character string operation.
- In some languages, pattern matching is supported directly in the language. In others, it is provided by a function or class library.

- **Character String Type in Certain Languages**
- If strings are not defined as a primitive type, string data is usually stored in **arrays of single characters** and referenced as such in the language.
- C and C++ use char arrays to store character strings.
- Many uses of strings and library functions use the convention that character strings are terminated with a special character, null, which is represented with zero. This is an alternative to maintaining the length of string variables.
- SNOBOL4 (a string manipulation language)

- 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

- There are **several design choices regarding the length of string values.**
- First, the **length can be static and set when the string is created**. Such a string is called a **static length string**.
- **Static**: COBOL, Java's String class
- The second option is to **allow strings to have varying length up to a declared and fixed maximum set by the variable's definition**, as exemplified by the strings in C and the C-style strings of C++.
- These are called **limited dynamic length strings**

- **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
- The third option is to allow strings to have varying length with no maximum, as in JavaScript, Perl, and the standard C++ library. These are called **dynamic length strings.**
- **Dynamic** (no maximum): SNOBOL4, Perl, JavaScript
- **Ada supports all three string length options**

- **Evaluation**
- String types are important to the writability of a language.
- As a primitive type with static length, they are inexpensive to provide
- Dynamic length is nice, but is it worth the expense?
- **Implementation of Character String Types**

Implementation of Character String Types

- A **descriptor** for a static character string type, which is required only during compilation, has three fields.
- The first field of every descriptor is the **name** of the type
- The second field is the type's **length** (in characters).
- The third field is the **address of the first character.**
- **Limited dynamic strings** require a **run-time descriptor** to store both the **fixed maximum length and the current length**

Figure 6.2

Compile-time descriptor for static strings

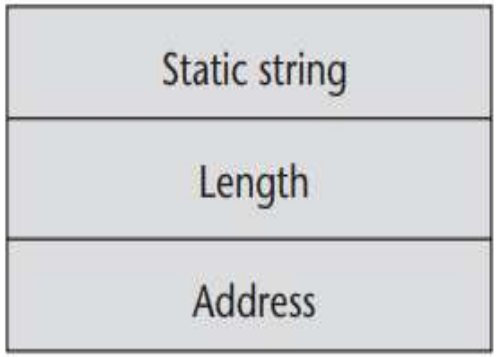
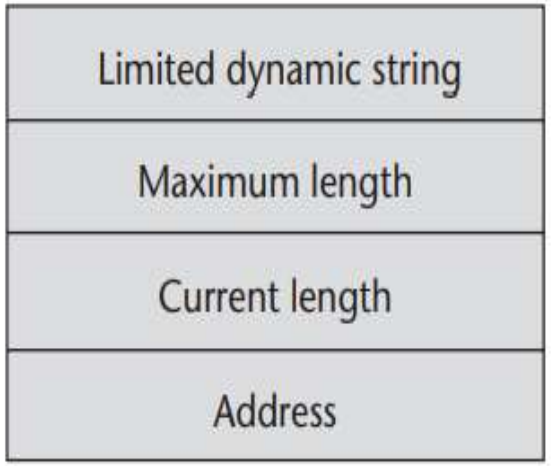


Figure 6.3

Run-time descriptor for limited dynamic strings



- **Dynamic length strings** require a simpler run-time descriptor because only the current length needs to be stored.
- Although we depict descriptors as independent blocks of storage, in most cases, they are stored in the **symbol table**

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
- In some languages, users can define two kinds of ordinal types: **enumeration** and **subrange**.

Enumeration Types

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

```
enum colors {red, blue, green, yellow, black};
colors myColor = blue, yourColor = red;
myColor++;                                  // would assign green to myColor
```

  - The enumeration constants are typically implicitly assigned the integer values, 0, 1, ..., but can be explicitly assigned any integer literal.

The design issues for enumeration types are as follows:

- 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 in the program checked?
- Are enumeration values coerced to integer?
- Are any other types 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

- 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 an aggregate of homogeneous 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?
- What is the maximum number of subscripts?
- Can array objects be initialized?
- Are any kind of slices supported?

Arrays and Indices

Array Indexing

- *Indexing* (or subscripting) is a mapping from indices to elements
  array_name (index_value_list)→an element
- Index Syntax
  - FORTRAN, PL/I, 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

Subscript Bindings and Array Categories

- There are five categories of arrays, based on the binding to subscript ranges, the binding to storage, and from where the storage is allocated
- 1. Static array
  2. Fixed stack-dynamic array
  3. Stack dynamic array
  4. Fixed heap dynamic array
  5. Heap dynamic array

Static array

- A static array is one in which the subscript ranges are statically bound and storage allocation is static (done before run time).
- The **advantage** of static arrays is efficiency:
- No dynamic allocation or deallocation is required.
- The **disadvantage** is that the storage for the array is fixed for the entire execution time of the program
- Int a[10];

Fixed stack-dynamic array

- A fixed stack-dynamic array is one in which the subscript ranges are statically bound, but the allocation is done at declaration elaboration time during execution.
- The advantage of fixed stack-dynamic arrays over static arrays is **space efficiency**
- You know the size of your array at compile time, but allow it to be allocated automatically on the stack (the size is fixed at compile time but the storage is allocated when you enter its scope, and released when you leave it)

- A large array in one subprogram can use the same space as a large array in a different subprogram, as long as both subprograms are not active at the same time.
- The same is true if the two arrays are in different blocks that are not active at the same time.
- The disadvantage is the required allocation and deallocation time

```
void foo()
{
  int fixed_stack_dynamic_array[7];
  /* ... */
}
```

Stack dynamic array

- A stack-dynamic array is one in which both the subscript ranges and the storage allocation are dynamically bound at elaboration time.
- Once the subscript ranges are bound and the storage is allocated, however, they remain fixed during the lifetime of the variable.
- The **advantage** of stack-dynamic arrays over static and fixed stack-dynamic arrays is **flexibility**.
- **The size of an array need not be known until the array is about to be used.**
- You don't know the size until runtime.

```
void foo(int n)
{
  int stack_dynamic_array[n];
  /* ... */
}
```

Fixed heap-dynamic array

- A fixed heap-dynamic array is similar to a fixed stack-dynamic array, in that **the subscript ranges and the storage binding are both fixed after storage is allocated.**
- The differences are that both the subscript ranges and storage bindings are done when the user program requests them during execution, and the **storage is allocated from the heap**, rather than the stack.

Fixed heap-dynamic array

- The **advantage** of fixed heap-dynamic arrays is **flexibility**—the array's size always fits the problem.
- The **disadvantage** is allocation time from the heap, which is longer than allocation time from the stack

```
int * fixed_heap_dynamic_array = malloc(7 * sizeof(int));
```

Heap-dynamic array

- A heap-dynamic array is one in which the binding of subscript ranges and storage allocation is dynamic and can change any number of times during the array's lifetime.
- The **advantage** of heap-dynamic arrays over the others is **flexibility**:
- Arrays can grow and shrink during program execution as the need for space changes.
- The **disadvantage** is that **allocation and deallocation take longer and may happen many times during execution of the program.**

```
void foo(int n)
{
  int * heap_dynamic_array = malloc(n * sizeof(int));
}
```

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

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

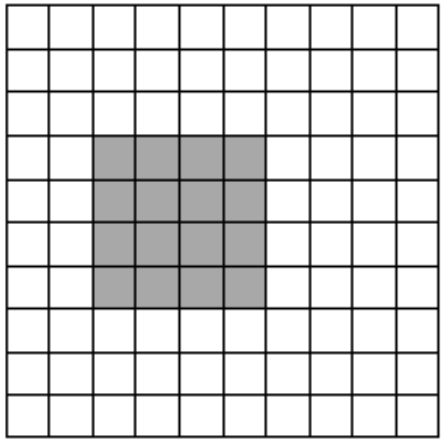
Rectangular and Jagged Arrays

- 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)

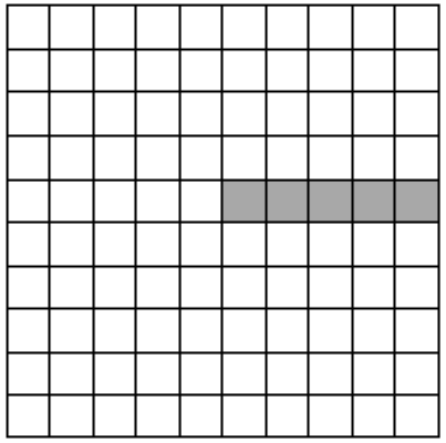
Slices

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

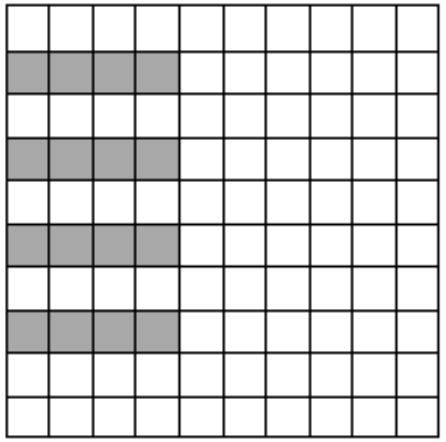
Slicing In Fortran 90



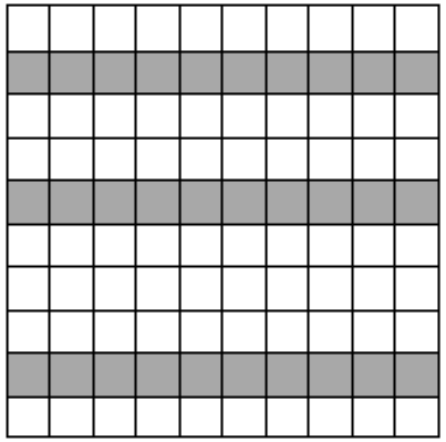
matrix(3:6, 4:7)



matrix(6:, 5)



matrix(:4, 2:8:2)



matrix(:, (/2, 5, 9/))

- **Perl** supports slices of two forms, a list of specific subscripts or a range of subscripts.
- For example, @list[1..5] = @list2[3, 5, 7, 9, 13];
- **Ruby** supports slices with the **slice** method of its **Array** object, which can take three forms of parameters.
  - A single integer expression parameter is interpreted as a subscript, in which case slice returns the element with the given subscript.

- If slice is given two integer expression parameters, the first is interpreted as a beginning subscript and the second is interpreted as the number of elements in the slice. For example, suppose list is defined as follows:
  - list = [2, 4, 6, 8, 10]
  - **list.slice(2, 2)** returns [6, 8]
- The third parameter form for slice is a **range**, which has the form of an integer expression, two periods, and a second integer expression.
- With a range parameter, slice returns an array of the element with the given range of subscripts.
- For example, **list.slice(1..3)** returns [4, 6, 8].

Implementation of Array Types

- Implementing arrays requires considerably **more compile-time effort** than does implementing primitive types.
- **The code to allow accessing of array elements must be generated at compile time.**
- At run time, this code must be executed to produce element addresses.

- 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

- For example, if the matrix had the values

  3 4 7
  6 2 5
  1 3 8

  - it would be stored in row major order as:

    3, 4, 7, 6, 2, 5, 1, 3, 8

  - If the example matrix above were stored in column major, it would have the following order in memory.

    3, 6, 1, 4, 2, 3, 7, 5, 8

- In all cases, sequential access to matrix elements will be faster if they are accessed in the order in which they are stored, because that will minimize the **paging**. (Paging is the movement of blocks of information between disk and main memory. The objective of paging is to keep the frequently needed parts of the program in memory and the rest on disk.)

- Locating an Element in a Multi-dimensioned Array (row major)

Location (a[i,j]) = address of a [1,1] + ((i - 1) * n + (j - 1)) * element_size

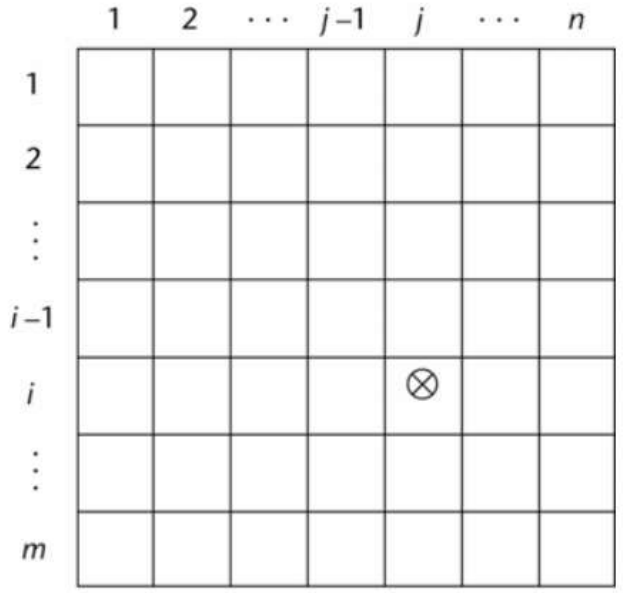


Figure 6.4

Compile-time descriptor for single-dimensioned arrays

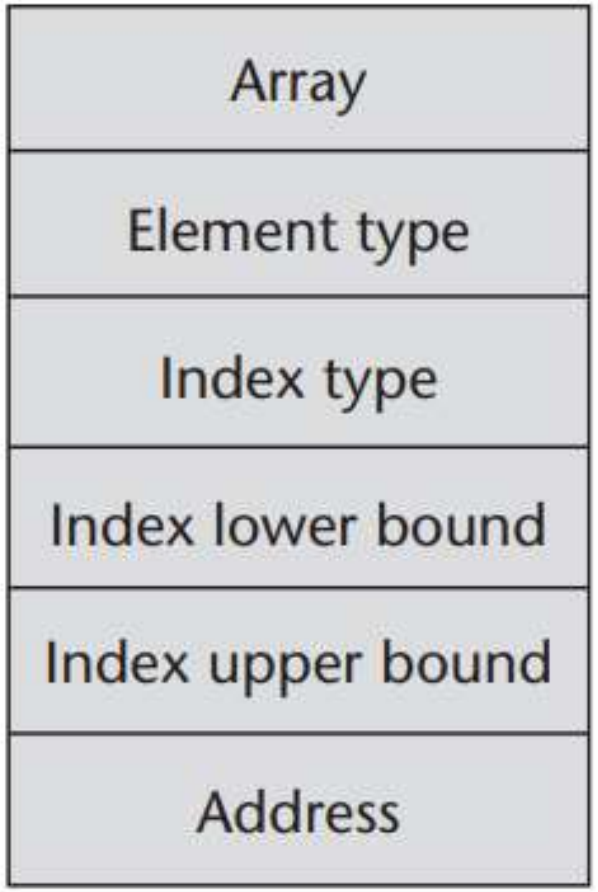
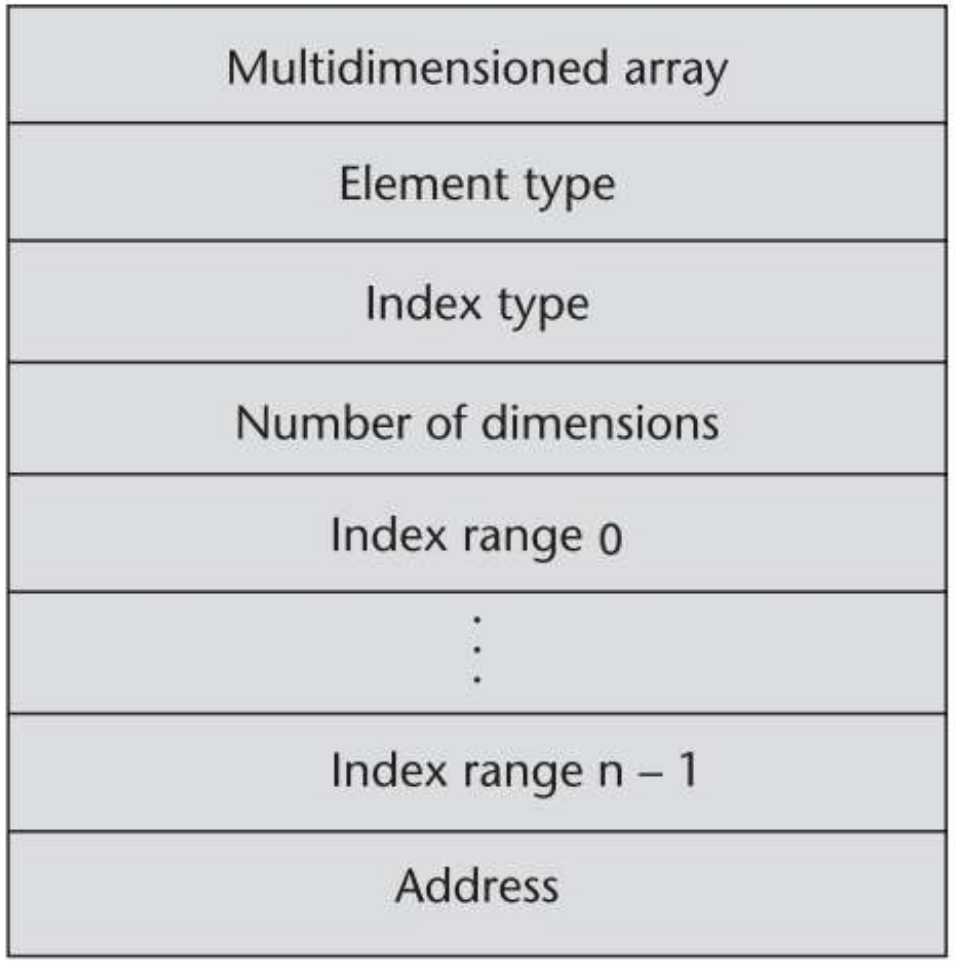


Figure 6.6

A compile-time descriptor for a 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**.
- So each element of an associative array is in fact a pair of entities, a key and a value.
- Associative arrays are supported by the standard class libraries of Java and C++ and Perl.
- Example: In Perl, associative arrays are often called **hashes**. Names begin with %; literals are delimited by parentheses

```
%temps = ("Mon" => 77, "Tue" => 79, “Wed” => 65);
```

- Each hash element consists of two parts: **a key**, which is a string, and a **value**, which is a scalar (number, string, or reference).
- Hashes can be set to literal values with the assignment statement, as in
- %salaries = ("Gary" => 75000, "Perry" => 57000, "Mary" => 55750, "Cedric" => 47850);

- Subscripting is done using braces and keys

  $temps{"Wed"} = 83;

- Elements can be removed with delete

  delete $temps{"Tue"};

- Elements can be emptied by assigning the empty literal

  @temps = ( );

- The **exists operator** returns true or false, depending on whether its operand key is an element in the hash.
- if (exists $salaries{"Shelly"})
- The **keys operator**, when applied to a hash, returns an array of the keys of the hash.
- The **values operator** does the same for the values of the hash.
- The **each operator** iterates over the element pairs of a hash.

Loop Through an Associative Array

```php
<?php
$age = array("Peter"=>"35", "Ben"=>"37", "Joe"=>"43");

foreach($age as $x => $x_value) {
  echo "Key=" . $x . ", Value=" . $x_value;
  echo "<br>";
}
?>
```

Record Types

- A record is a possibly **heterogeneous** aggregate of data elements in which the individual elements are identified by names.
- In C, C++, and C#, records are supported with the **struct** data type. In C++, structures are a minor variation on classes.
- COBOL uses level numbers to show nested records; others use recursive definition
- 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 Employee_Name_Type is record
   First : String (1..20);
   Middle : String (1..10);
   Last : String (1..20);
end record;
type Employee_Record_Type is record
   Employee_Name: Employee_Name_Type;
   Hourly_Rate: Float;
end record;
Employee_Record: Employee_Record_Type;
```

- The following design issues are specific to records:
  - What is the syntactic form of references to fields ?
  - Are elliptical references allowed ?

- **Fully qualified reference** to a record field is one in which all intermediate record names, from the largest enclosing record to the specific field, are named in the reference.
- Both the COBOL and the Ada example field references above are fully qualified.
- COBOL allows elliptical references to record fields.
- In an **elliptical reference**, the field is named, but any or all of the enclosing record names can be omitted, as long as the resulting reference is unambiguous in the referencing environment.

References to Records

- Most language use dot notation

  Emp_Rec.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

For example, the MIDDLE field in the COBOL record example above can be referenced with

MIDDLE OF EMPLOYEE-NAME OF EMPLOYEE-RECORD

Most of the other languages use **dot notation** for field references

Employee_Record.Employee_Name.Middle

Operations on Records

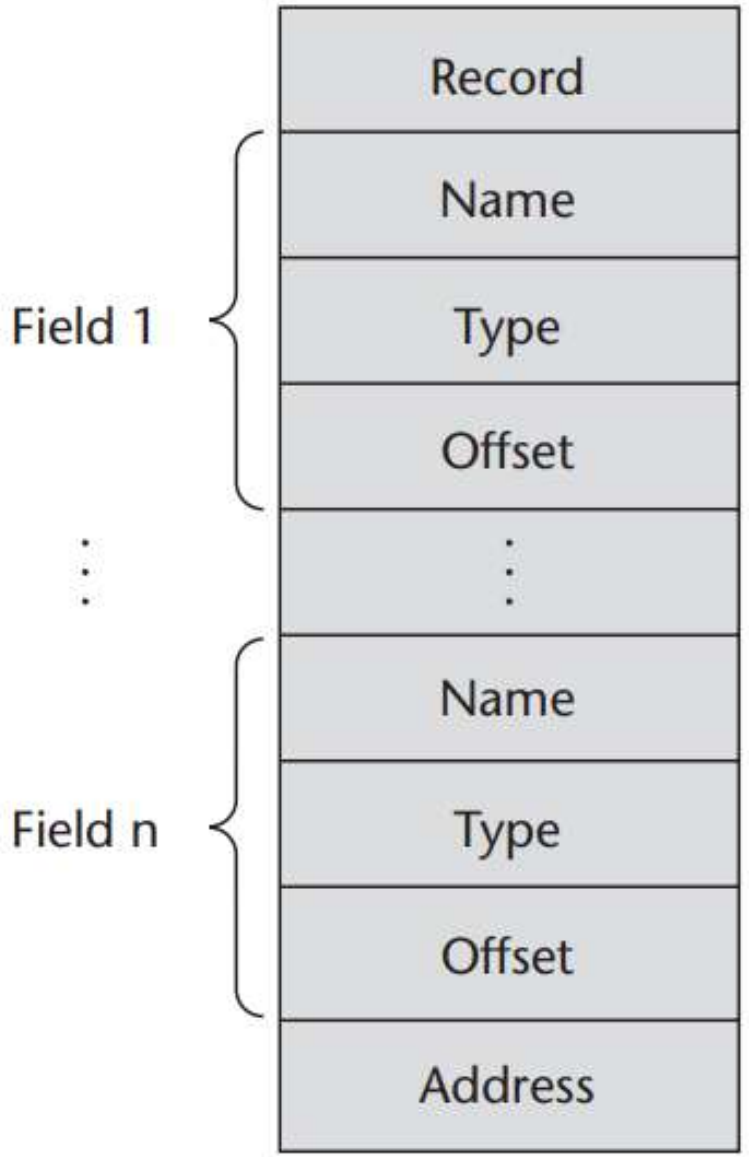
- Assignment is very common if the types are identical
- Ada allows record comparison
- Ada records can be initialized with aggregate literals
- COBOL provides MOVE CORRESPONDING
    - Copies a field of the source record to the corresponding field in the target record

Evaluation

- Records are used when the collection of data values is heterogeneous and the different fields are not processed in the same way.
- Also, the fields of a record often need not be processed in a particular order.
- Field names are like literal, or constant, subscripts. Because they are static, they provide very efficient access to the fields.
- Dynamic subscripts could be used to access record fields, but it would disallow type checking and would also be slower

Figure 6.7

A compile-time descriptor for a record



Union

- **A union is a type whose variables may store different type values at different times during program execution**
- C and C++ provide union constructs in which there is no language support for type checking.
- In C and C++, the union construct is used to specify union structures.
- The unions in these languages are called free unions, because programmers are allowed complete freedom from type checking in their use.

```
union flexType {
    int intEl;
    float floatEl;
};
union flexType el1;
float x;
...
el1.intEl = 27;
x = el1.floatEl;
```

This last assignment is not type checked, because the system cannot determine the current type of the current value of `el1`, so it assigns the bit string representation of `27` to the **`float`** variable `x`, which of course is nonsense.

- Type checking of unions requires that each union construct include a **type indicator**.
- Such an indicator is called a **tag**, or discriminant, and a union with a discriminant is called a **discriminated union**.
- The first language to provide discriminated unions was ALGOL 68. They are now supported by Ada, ML, Haskell, and F#

Unions Types

- A union is a type whose variables are allowed to store different type values at different times during execution.

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 *discriminanted union.*
- Supported by **Ada**

Ada Union Types

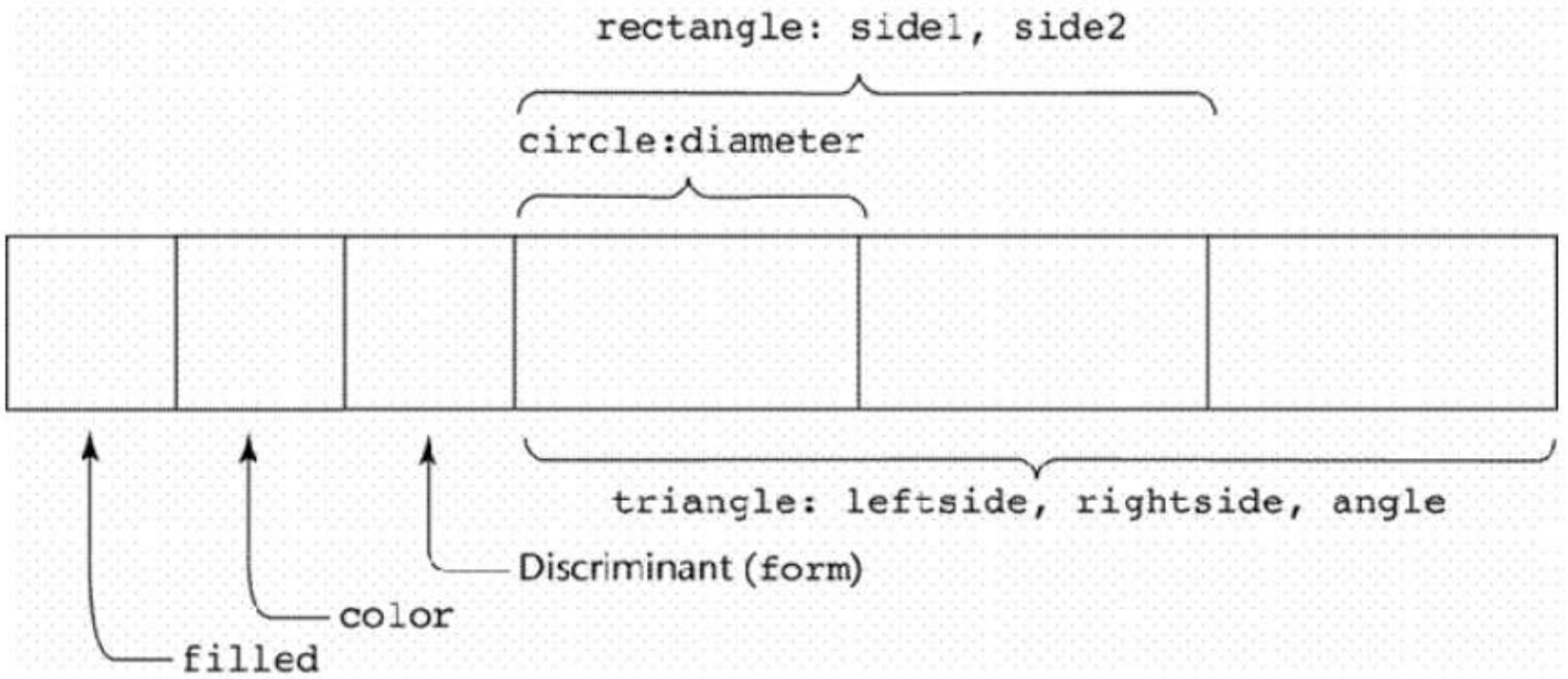
```
type Shape is (Circle, Triangle, Rectangle);
type Colors is (Red, Green, Blue);
type Figure (Form: Shape) is record
    Filled: Boolean;
    Color: Colors;
    case Form is
        when Circle => Diameter: Float;
        when Triangle =>
            Leftside, Rightside: Integer;
            Angle: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

The structure of this variant record is shown in Figure 6.8. The following two statements declare variables of type `Figure`:

```
Figure_1 : Figure;
Figure_2 : Figure(Form => Triangle);
```

`Figure_1` is declared to be an unconstrained variant record that has no initial value. Its type can change by assignment of a whole record, including the discriminant, as in the following:

```
Figure_1 := (Filled => True,
             Color => Blue,
             Form => Rectangle,
             Side_1 => 12,
             Side_2 => 3);
```



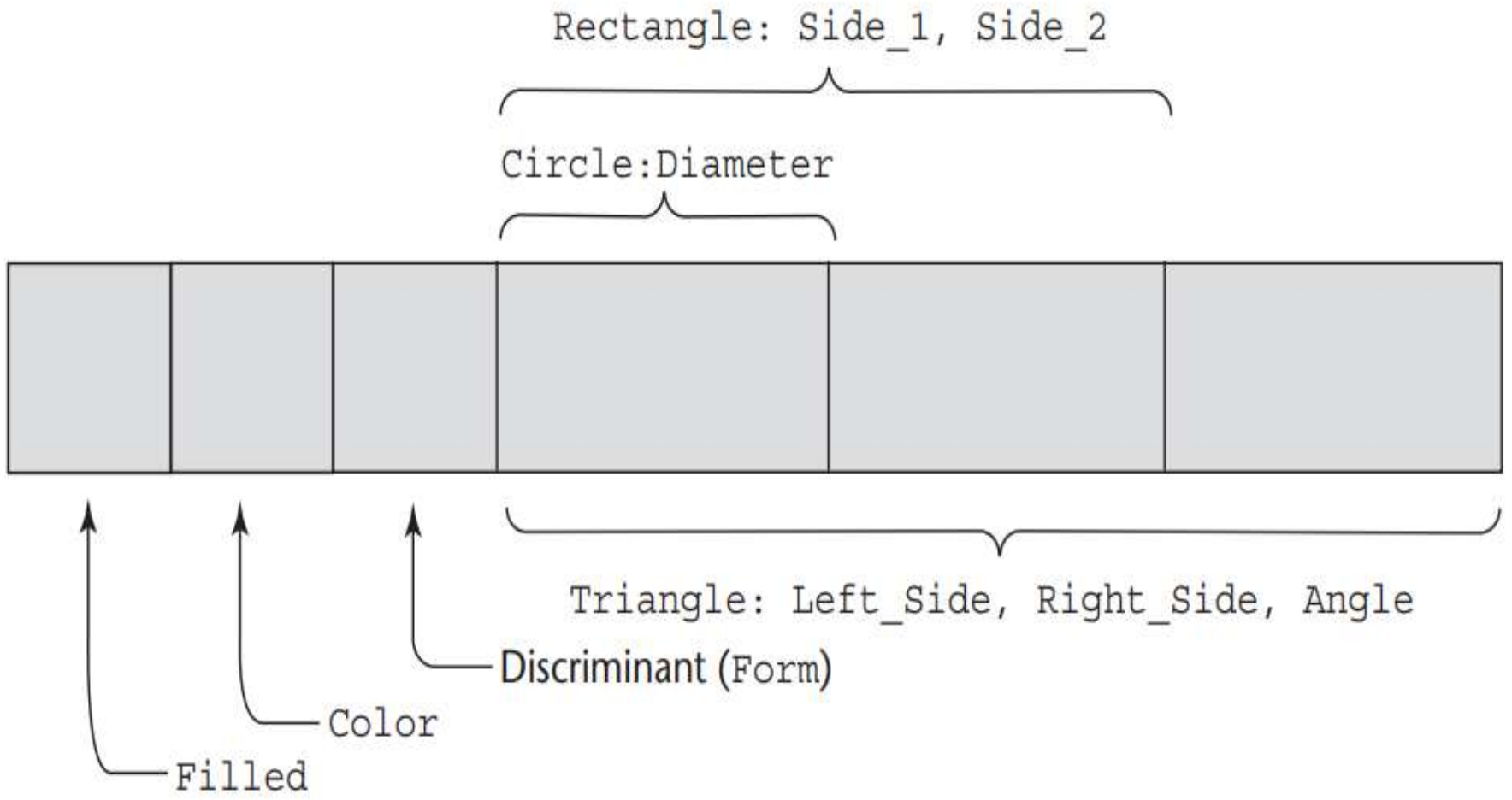
Ada Union Type Illustrated: A discriminated union of three shape variables

Evaluation of Unions

- Potentially **unsafe** construct
- Java and C# **do not** support unions
- Reflective of growing concerns for safety in programming language

Figure 6.8

A discriminated union of three shape variables (assume all variables are the same size)



Evaluation

- Unions are potentially unsafe constructs in some languages.
- They are one of the reasons why C and C++ are not strongly typed:
- These languages do not allow type checking of references to their unions.
- On the other hand, unions can be safely used, as in their design in Ada, ML, Haskell, and F#.

Implementation of Union Types

- Unions are implemented by simply using the same address for every possible variant.
- Sufficient storage for the largest variant is allocated.
- The tag of a discriminated union is stored with the variant in a recordlike structure.

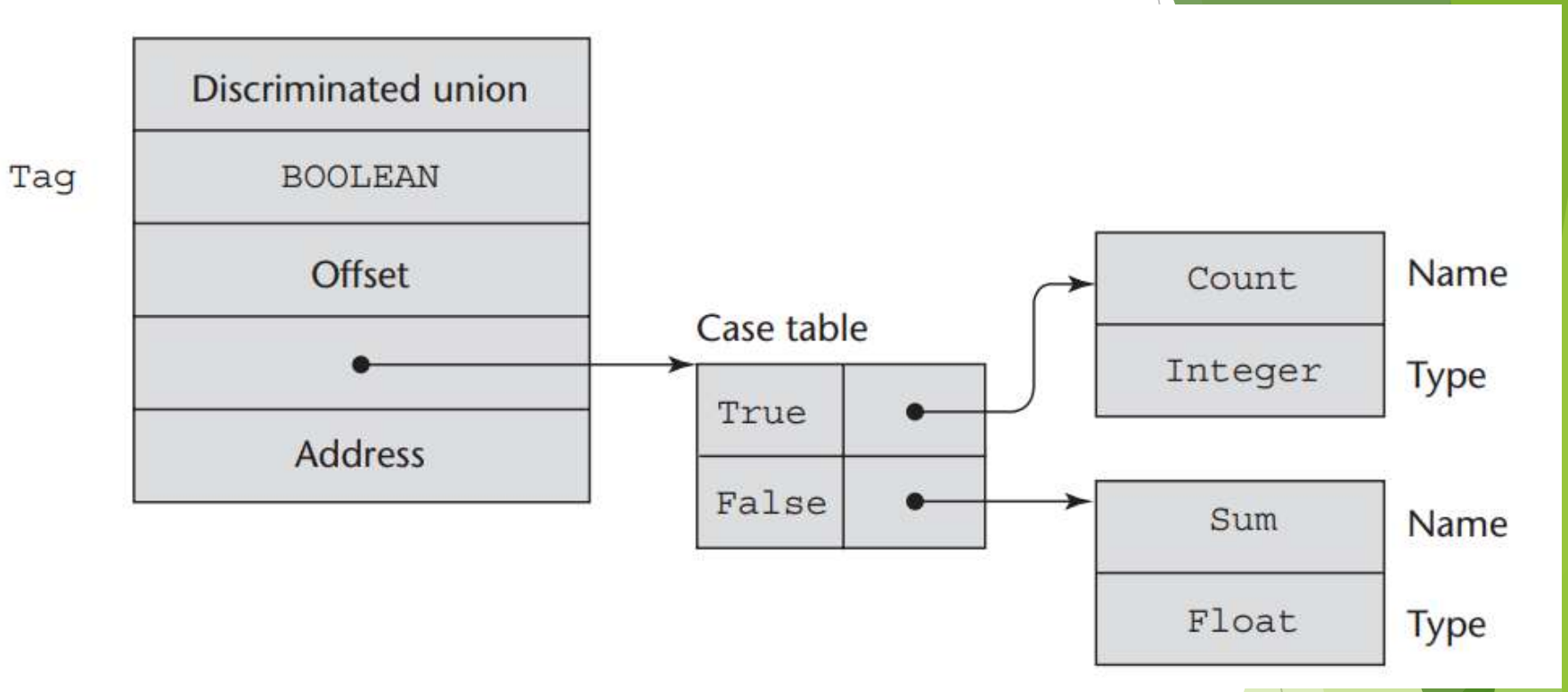
Ada example:

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

The descriptor for this type could have the form shown in Figure 6.9.

Figure 6.9

A compile-time descriptor for a discriminated union



List Types

- Lists were first supported in the first functional programming language, LISP.
- They have always been part of the functional languages, but in recent years they have found their way into some imperative languages.
- Lists in Scheme and Common LISP are delimited by parentheses and the elements are not separated by any punctuation.
- For example
- **(A B C D)**
- Nested lists have the same form,
- so we could have **(A (B C) D)**

- Data and code have the same syntactic form in LISP and its descendants.
- If the list (A B C) is interpreted as code, it is a call to the function A with parameters B and C.
- The fundamental list operations in Scheme are two functions that take lists apart and two that build lists.
- The **CAR function** returns the first element of its list parameter.
- For example, consider the following example:
- **(CAR '(A B C)) - Returns A**
- The quote before the parameter list is to prevent the interpreter from considering the list a call to the A function with the parameters B and C, in which case it would interpret it.

- The **CDR function** returns its parameter list minus its first element. For example,
- Consider the following example:
- **(CDR '(A B C))**
- This function call returns the **list (B C)**

- Common LISP also has the functions FIRST (same as CAR), SECOND, . . . , TENTH, which return the element of their list parameters that is specified by their names

- In Scheme and Common LISP, **new lists are constructed with the CONS and LIST functions.**
- The function CONS takes two parameters and returns a new list with its first parameter as the first element and its second parameter as the remainder of that list.
- For example, consider the following
- **(CONS 'A '(B C)) – This call returns the new list (A B C).**

- The **LIST function** takes any number of parameters and returns a new list with the parameters as its elements.
- For example, consider the following call to LIST:
- **(LIST 'A 'B '(C D))**
- This call returns the new list **(A B (C D)).**

- **ML** has lists and list operations, although their appearance is not like those of Scheme.
- Lists are specified in square brackets, with the elements separated by commas, as in the following list of integers: [5, 7, 9]

- [] is the empty list, which could also be specified with nil.
- The Scheme CONS function is implemented as a binary infix operator in ML, represented as ::
- For example, **3 :: [5, 7, 9]** returns the following new list: **[3, 5, 7, 9]**
- The elements of a list must be of the same type, so the following list would be illegal: [5, 7.3, 9]

- ML has functions that correspond to Scheme's CAR and CDR, named **hd** (head) and **tl** (tail).
- For example,
- hd [5, 7, 9] is 5
- tl [5, 7, 9] is [7, 9]

- **The lists of Python are mutable.**
- They can contain any data value or object.
- A Python list is created with an assignment of a list value to a name.
- **A list value is a sequence of expressions that are separated by commas and delimited with brackets.**
- For example, consider the following statement:
- myList = [3, 5.8, "grape"]
- **The elements of a list are referenced with subscripts in brackets**, as in the following example:
- x = myList[1] This statement assigns 5.8 to x.

- **The elements of a list are indexed starting at zero.**
- List elements also can be updated by assignment.
- A list element can be deleted with **del**, as in the following statement:
- del myList[1]
- This statement removes the second element of myList.

List Comprehensions

- Python includes a powerful mechanism for creating arrays called **list comprehensions.**
- The mechanics of a list comprehension is that **a function is applied to each of the elements of a given array and a new array is constructed from the results.**
- The syntax of a Python list comprehension is as follows
- [expression **for** iterate_var **in** array **if** condition]
- Consider the example: [x * x for x in range(12) if x % 3 == 0]
- The range function creates the array
  - [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12].

- The conditional filters out all numbers in the array that are not evenly divisible by 3.
- Then, the expression squares the remaining numbers.
- The results of the squaring are collected in an array, which is returned.
- This list comprehension returns the following array:
- **[0, 9, 36, 81]**

Slices of lists are also supported in Python.

Haskell's list comprehensions have the following form:

[body | qualifiers]

For example, consider the following definition of a list:

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

This defines a list of the squares of the numbers from 1 to 10.

F# includes list comprehensions, which in that language can also be used to create arrays. For example, consider the following statement:

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

This statement creates the array [1; 4; 9; 16; 25] and names it myArray.

Pointer $ Reference Types

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 Illustration

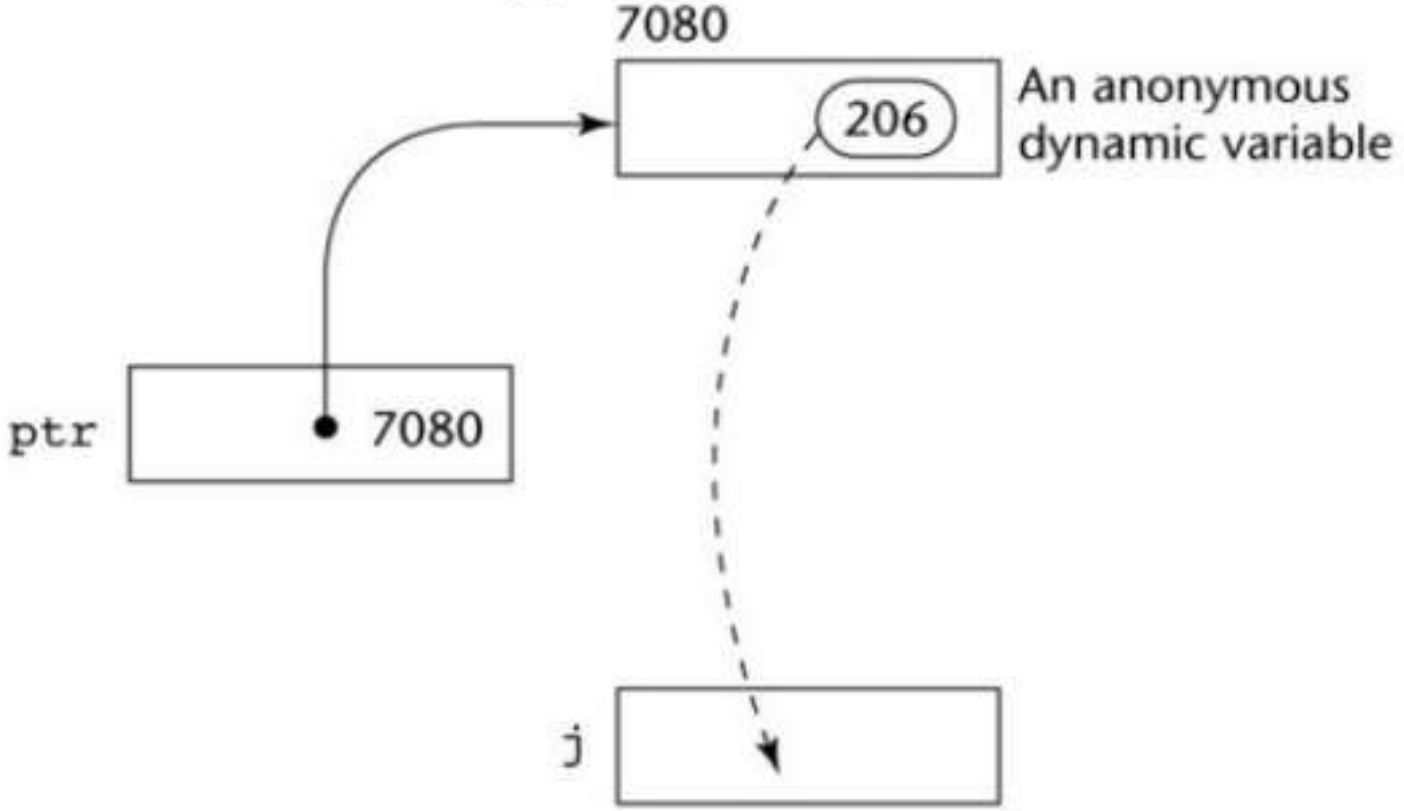


Figure 3.9 The assignment operation j = *ptr

Dangling Pointers

The following sequence of operations creates a dangling pointer in many languages:

1. A new heap-dynamic variable is created and pointer `p1` is set to point at it.
2. Pointer `p2` is assigned `p1`'s value.
3. The heap-dynamic variable pointed to by `p1` is explicitly deallocated (possibly setting `p1` to `nil`), but `p2` is not changed by the operation. `p2` is now a dangling pointer. If the deallocation operation did not change `p1`, both `p1` and `p2` would be dangling. (Of course, this is a problem of aliasing—`p1` and `p2` are aliases.)

Lost Heap-Dynamic Variables

- A **lost heap-dynamic variable** is an allocated heap-dynamic variable that is no longer accessible to the user program.
- Such variables are often called **garbage**, because they are not useful for their original purpose, and they also cannot be reallocated for some new use in the program.
- **Lost heap-dynamic variables** are most often created by the following sequence of operations:
- 1. Pointer p1 is set to point to a newly created heap-dynamic variable.
- 2. p1 is later set to point to another newly created heap-dynamic variable.
- The first heap-dynamic variable is now inaccessible, or lost. This is sometimes called **memory leakage.**
- Memory leakage is a problem, regardless of whether the language uses implicit or explicit deallocation

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)

Solutions to the Dangling-Pointer Problem

- **Tombstones**
- Every heap-dynamic variable includes a special cell, called a tombstone, that is itself a pointer to the heap-dynamic variable.
- The actual pointer variable points only at tombstones and never to heap-dynamic variables.
- When a heap-dynamic variable is deallocated, the tombstone remains but is set to nil, indicating that the heap-dynamic variable no longer exists.
- This approach prevents a pointer from ever pointing to a deallocated variable.
- Any reference to any pointer that points to a nil tombstone can be detected as an error.

- Tombstones are **costly** in both time and space. Because tombstones are never deallocated, their storage is never reclaimed.
- Every access to a heap dynamic variable through a tombstone **requires one more level of indirection**, which requires an additional machine cycle on most computers.
- Apparently none of the designers of the more popular languages have found the additional safety to be worth this additional cost, because **no widely used language uses tombstone.**

Locks-and-keys approach

- **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
- **Every access to the dereferenced pointer compares the key value of the pointer to the lock value in the heap-dynamic variable.**
- If they match, the access is legal; otherwise the access is treated as a run-time error.
- Any copies of the pointer value to other pointers must copy the key value.

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.**
- As a result, although it is natural to perform arithmetic on addresses, **it is not sensible to do arithmetic on references.**

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++

Heap Management

- Garbage Collection
- Language implementation notice when objects are no longer useful and reclaim them automatically

- 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 counters

- Maintains in every cell a **counter** that stores the number of pointers that are currently pointing at the cell.
- Embedded in the decrement operation for the reference counters, which occurs when a pointer is disconnected from the cell, is a check for a zero value.
- If the reference counter reaches zero, it means that no program pointers are pointing at the cell, and it has thus become garbage and can be returned to the list of available space.

Drawbacks

- There are three distinct problems with the reference counter method:

1. If storage cells are relatively small, the space required for the counters is significant.
2. Some execution time is obviously required to maintain the counter values. Every time a pointer value is changed, the cell to which it was pointing must have its counter decremented, and the cell to which it is now pointing must have its counter incremented.
3. Complications arise when a collection of cells is connected circularly. The problem here is that each cell in the circular list has a reference counter value of at least 1, which prevents it from being collected and placed back on the list of available space

Advantage

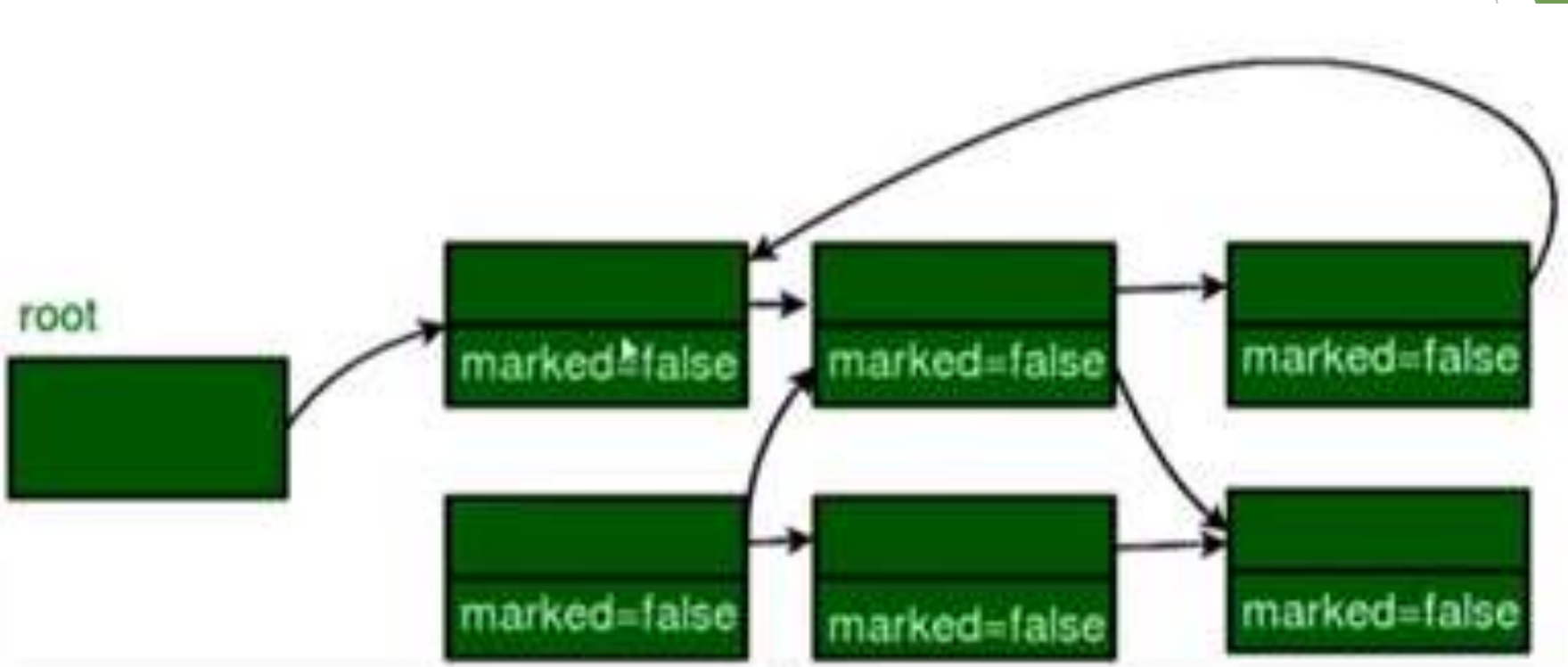
- Its actions are interleaved with those of the application, so it never causes significant delays in the execution of the application.

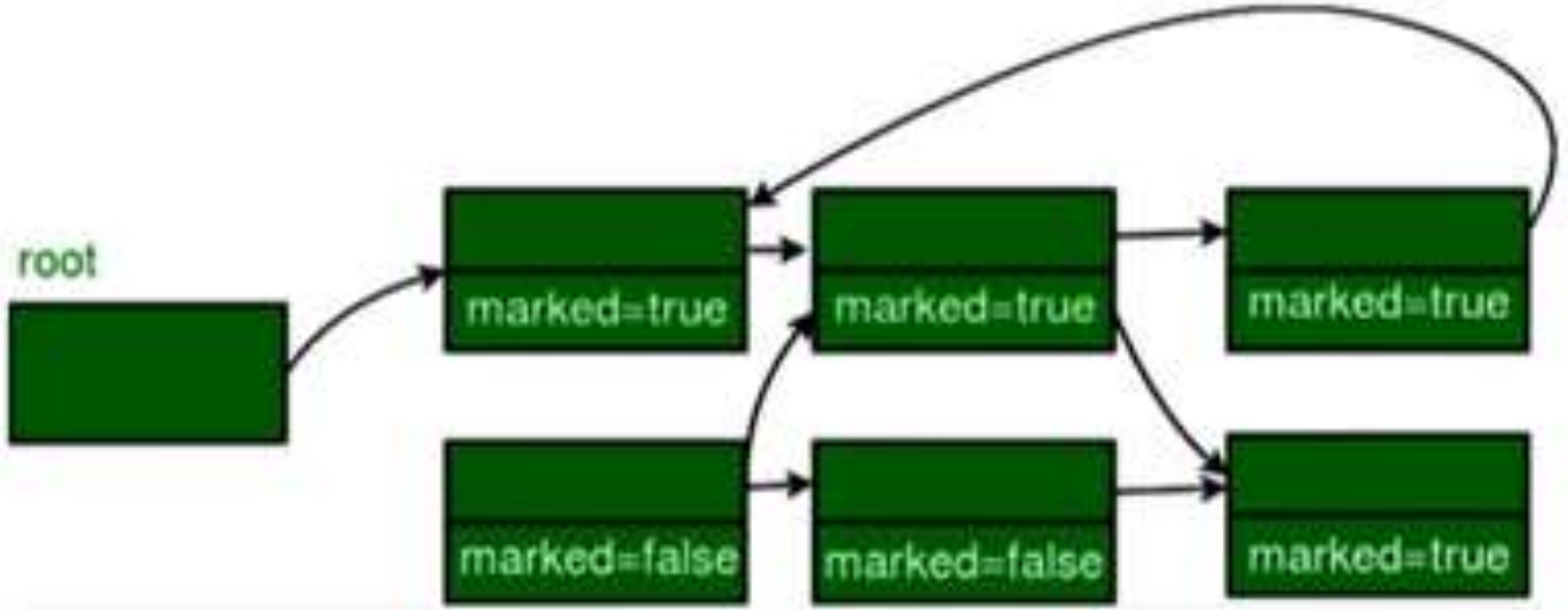
Mark & Sweep

- Executed when the amount of free space in heap falls below some threshold
- 3 steps are executed by garbage collector

1. Collector walks through the heap, marking every block as useless
2. Collector recursively explores all linked data structures in the program marking each newly discovered block as useful
3. Collector again walks through the heap, moving every lock that is still marked useless to free list

Mark and sweep




root
marked=true
marked=true
marked=true
marked=false
marked=false
marked=true

Disadvantages

- In its original form, it was **done too infrequently**.
- When done, it caused **significant delays in application execution.**
- Contemporary marksweep algorithms avoid this by doing it more often—called **incremental marksweep.**

Type checking

- Type checking is the activity of ensuring that the **operands of an operator are of compatible types.**
- A compatible type is one that either is legal for the operator or is allowed under language rules to be implicitly converted by compiler-generated code (or the interpreter) to a legal type. This **automatic conversion** is called a **coercion**.
- Eg: If an int variable and a float variable are added in Java, the value of the int variable is coerced to float and a floating-point addition is done.

- A **type error** is the application of an operator to an operand of an inappropriate type.
- Eg: In the original version of C, if an int value was passed to a function that expected a float value, a type error would occur
- If all bindings of variables to types are static in a language, then type checking can nearly always be done statically.
- Dynamic type binding requires type checking at run time, which is called **dynamic type checking**.
- **A programming language is strongly typed if type errors are always detected.**

- Ada is nearly strongly typed.
- It is only nearly strongly typed because it allows programmers to breach the type-checking rules by specifically requesting that type checking be suspended for a particular type conversion.
- It take place by extracting the value of a variable of one type and using it as if it were of a different type. This kind of conversion is sometimes called a **nonconverting cast**

Type Equivalence

- The type compatibility rules are simple and rigid for the predefined scalar types.
- However, in the cases of structured types, such as arrays and records and some user-defined types, the rules are more complex.
- Coercion of these types is rare, so the issue is not type compatibility, but type equivalence
- Two types are equivalent if an operand of one type in an expression is substituted for one of the other type, without coercion. Type equivalence is a strict form of type compatibility—compatibility without coercion.

Type Equivalence

- Two types are considered same if they have

1. Name Equivalence
2. Structural Equivalence

Name Equivalence

- Name Equivalence - Two types are equal if and only if they have **same type name**

```
typedef strct{
        int data[100];
        int count;
        }stack;

typedef strct{
        int data[100];
        int count;
                }set;
```

- Stack x,y;
- Set r,s;

- Using name equivalance
- x and y are of same type and r and s are of same type
- But type of x and y will not be equivalent to r and s
- i.e. X=y and r=s are valid but x=r is invalid

Structural Equivalence

- Two types are equal if and only if they have same structure.
- i.e name and types of each component must be same and must be listed in same order in type definition
- Here 2 types stack and set can be considered equivalent i.e x=r is valid

- Ada follows name equivalence and ML follows structural equivalence

Expressions and Assignment Statements

Introduction

- Expressions are the fundamental means of specifying computations in a programming language.
- To understand expression evaluation, need to be familiar with the orders of operator and operand evaluation.
- Essence 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).
- The purpose of an arithmetic expression is to specify an arithmetic computation.
- An implementation of such a computation must cause two actions:
    - Fetching the operands from memory
    - 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**
   - 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).
   - 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.
- Unary addition **(+)** is called the **identity operator** because it usually has no associated operation and thus has no effect on its operand.
- 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.
- 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:
  - Left to right, except **, which is right to left
  - 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
```

Exponentiation in Fortran and Ruby is right associative, so in the expression

```
A ** B ** C
```

the right operator is evaluated first.

In Ada, exponentiation is nonassociative, which means that the expression

```
A ** B ** C
```

is illegal. Such an expression must be parenthesized to show the desired order, as in either

```
(A ** B) ** C
```

or

```
A ** (B ** C)
```

In Visual Basic, the exponentiation operator, ^, is left associative. The associativity rules for a few common languages are given here:

Language	*Associativity Rule*
Ruby	Left: *, /, +, -
	Right: **
C-based languages	Left: *, /, %, binary +, binary -
	Right: ++, --, unary -, unary +
Ada	Left: all except **
	Nonassociative: **

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;

Ruby Expressions

- Recall that Ruby is a pure object-oriented language, which means, among other things, that every data value, including literals, is an object.
- Ruby supports the collection of arithmetic and logic operations that are included in the C-based languages.
- What sets Ruby apart from the C-based languages in the area of expressions is that all of the arithmetic, relational, and assignment operators, as well as array indexing, shifts, and bitwise logic operators, are implemented as methods.
- For example, the expression a + b is a call to the + method of the object referenced by a, passing the object referenced by b as a parameter.

- One interesting result of the implementation of operators as methods is that they can be overridden by application programs. Therefore, these operators can be redefined

Expressions in LISP

- As is the case with Ruby, all arithmetic and logic operations in LISP are performed by subprograms.
- But in LISP, the subprograms must be explicitly called. For example, to specify the C expression a + b * c in LISP, one must write the following expression:
- (+ a (* b c))
- In this expression, + and * are the names of functions

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. If an operand is a parenthesized expression, all of the operators it contains must be evaluated before its value can be used as an operand.

Side Effects (IMP)

- A side effect of a function, naturally called a functional side effect, occurs when the function changes either one of its parameters or a global variable.
- (A global variable is declared outside the function but is accessible in the function.)
- Consider the expression **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.

- Consider the following situation: fun returns 10 and changes the value of its parameter to 20
- Suppose we have the following:
- a = 10; b = a + fun(a);
- Then, if the value of a is fetched first (in the expression evaluation process), its value is 10 and the value of the expression is 20.
- But if the second operand is evaluated first, then the value of the first operand is 20 and the value of the expression is 30.

The following C program illustrates the same problem when a function changes a global variable that appears in an expression:

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

The value computed for `a` in `main` depends on the order of evaluation of the operands in the expression `a + fun1()`. The value of `a` will be either `8` (if `a` is evaluated first) or `20` (if the function call is evaluated first).

Solutions

- Two possible solutions:
  1. Write the language definition to disallow functional side effects
     - No two-way parameters in functions.
     - No non-local references in functions.
     - **Advantage**: it works!
     - **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
     - **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**

Referential Transparency

- A program has the property of referential transparency if 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 of the program.
- The value of a referentially transparent function depends entirely on its parameters
- Eg:
- result1 = (fun(a) + b) / (fun(a) - c); temp = fun(a);
- result2 = (temp + b) / (temp - c);
- If the function fun has no side effects, result1 and result2 will be equal, because the expressions assigned to them are equivalent

- However, suppose fun has the side effect of adding 1 to either b or c.
- Then result1 would not be equal to result2. So, that side effect violates the referential transparency of the program in which the code appears.

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”.

- When sensibly used, user-defined operator overloading can aid readability.
- For example, if + and * are overloaded for a matrix abstract data type and A, B, C, and D are variables of that type,
- then A * B + C * D
- can be used instead of
- MatrixAdd(MatrixMult(A, B), MatrixMult(C, D)
- On the other hand, user-defined overloading can be harmful to readability.
- For one thing, nothing prevents a user from defining + to mean multiplication

- Another consideration is the process of building a software system from modules created by different groups.
- If the different groups overloaded the same operators in different ways, these differences would obviously need to be eliminated before putting the system together.

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

In the C-based languages, explicit type conversions are called **casts**. To specify a cast, the desired type is placed in parentheses just before the expression to be converted, as in

```
(int) angle
```

One of the reasons for the parentheses around the type name in these conversions is that the first of these languages, C, has several two-word type names, such as `long int`.

In ML and F#, the casts have the syntax of function calls. For example, in F# we could have the following:

```
float(sum)
```

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 is an operator that compares the values of its two operands.
- A relational expression has two operands and one relational operator.
- The value of a relational expression is Boolean, except when Boolean is not a type included in the language.
- The relational operators are often overloaded for a variety of 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.

Short Circuit Evaluation

- A **short-circuit evaluation** of an expression is one in which the result is determined **without** evaluating all of the operands and/or operators.
- Ex:

```
(13 * a) * (b/13 - 1) // is independent of the value
                      (b/13 - 1) if a = 0, because 0*x = 0.
```

- So when a = 0, there is no need to evaluate `(b/13 - 1)` or perform the second multiplication.
- However, this shortcut is not easily detected during execution, so it is never taken.
- The value of the Boolean expression:

```
(a >= 0) && (b < 10) //  is independent of the second
                         expression if a < 0, because(F && x)
                         is False for all the values of x.
```

- So when a < 0, there is no need to evaluate b, the constant 10, the second relational expression, or the && operation.
- Unlike the case of arithmetic expressions, this shortcut can be easily discovered during execution.
- Short-circuit evaluation exposes the potential problem of side effects in expressions

```
(a > b) || (b++ / 3) //  b is changed only when a <= b.
```

- If the programmer assumed b would change every time this expression is evaluated during execution, the program will fail.
- C, C++, and Java: use short-circuit evaluation for the usual Boolean operators (**&&** and **||**), but also provide bitwise Boolean operators that are not short circuit (**&** and **|**)

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.

```
sum += value; ⇔     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 stand-alone 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

$$a = b + (c = d / b{+}{+}) - 1$$

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).