UNIT-2
Data Types and Variables

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?

2.1 Primitive Data Types – CO2

- 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

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

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

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
- Literal form (in Python):
  (7 + 3j), where 7 is the real part and 3 is the imaginary part

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

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

Character

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

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

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?

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/de-allocation is the biggest implementation problem

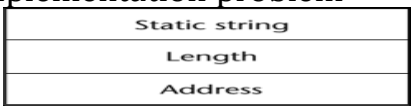


Figure 3.1 Compile-Time Descriptor

Figure 3.2 Run-Time Descriptors

2.2 User-Defined Ordinal Types - CO2

- 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

- 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?

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
- 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

Arrays Operations

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

Rectangular and Jagged Arrays

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

Slices

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

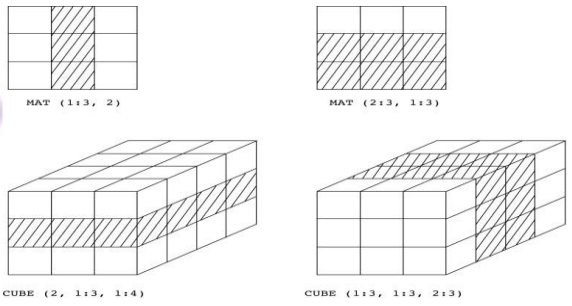
Slice Examples

- Fortran 95

```
Integer, Dimension (10) :: Vector
Integer, Dimension (3, 3) :: Mat
Integer, Dimension (3, 3) :: Cube
```

Vector (3:6) is a four element array

Figure 3.3 Slices Examples in Fortran 95



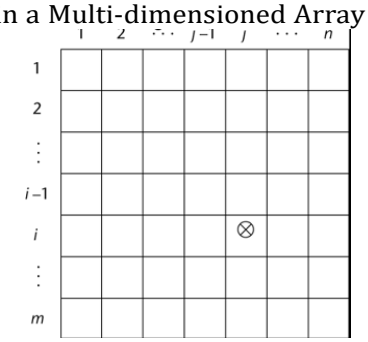
Implementation of Arrays

- 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

Figure 3.4 Locating an Element in a Multi-dimensioned Array



Compile-Time Descriptors

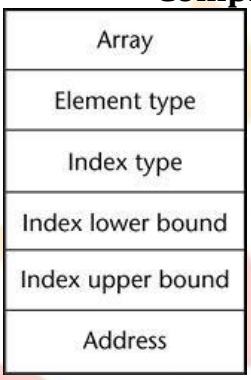


Figure 3.5 Single-Dimensioned Array

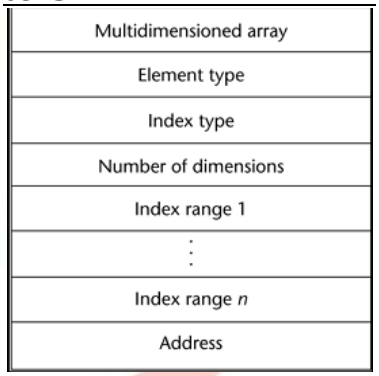


Figure 3.6 Multi-Dimensioned Array

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?

Associative Arrays in Perl

- Names begin with **%;** literals are delimited by parentheses

  %hi_temps = ("Mon" => 77, "Tue" => 79, —Wed|| => 65, …);
- Subscripting is done using braces and keys

  $hi_temps{"Wed"} = 83;
- Elements can be removed with delete

  delete $hi_temps{"Tue"};

2.3 Record Types – CO2

- 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
  - COBOL
    field_name OF record_name_1 OF … OF record_name_n
  - 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

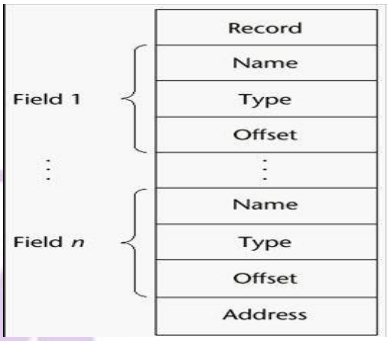


Figure 3.7 Implementation of Record Type

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

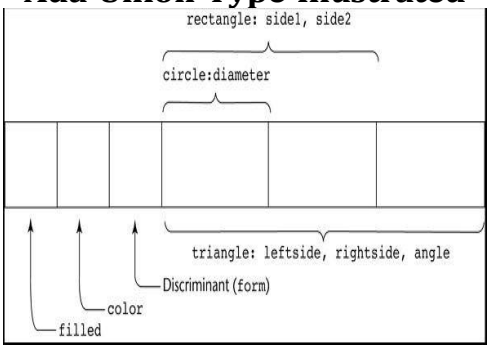


Figure 3.8 A Discriminated Union of Three Shape Variables

Evaluation of Unions

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

Pointer and Reference Types

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

Design Issues of Pointers

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

Pointer Operations

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

    *j = *ptr*

    sets j to the value located at ptr

Pointer Assignment Illustration

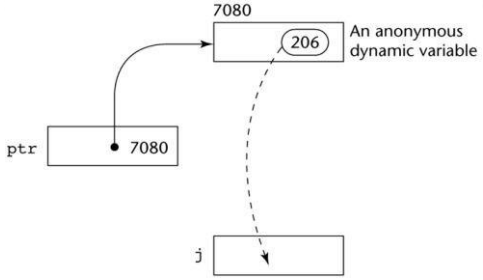


Figure 3.9 The assignment operation j = *ptr

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

- 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 marksweep algorithms avoid this by doing it more often—called incremental marksweep

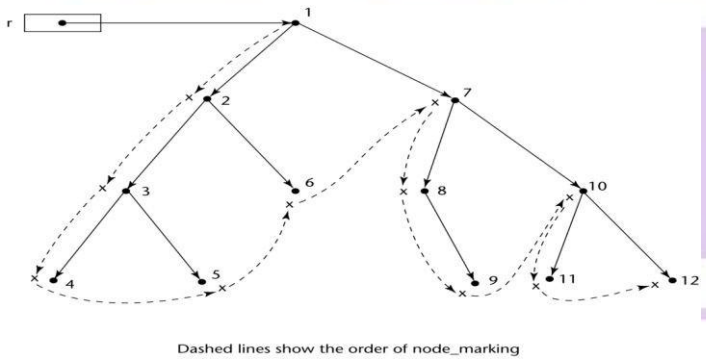


Figure 3.10 Marking Algorithm

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

2.4 Names – CO2

- Design issues for names:
  - Maximum length?
  - Are connector characters allowed?
  - Are names case sensitive?
  - Are special words reserved words or keywords?
- Length
  - If too short, they cannot be connotative
  - Language examples:
    - FORTRAN I: maximum 6
    - COBOL: maximum 30
    - FORTRAN 90 and ANSI C: maximum 31
    - Ada and Java: no limit, and all are significant
    - C++: no limit, but implementors often impose one
- Connectors
  - Pascal, Modula-2, and FORTRAN 77 don't allow
  - Others do
- Case sensitivity
  - Disadvantage: readability (names that look alike are different)
    - worse in C++ and Java because predefined names are mixed case (**e.g.**, IndexOutOfBoundsException)
  - C, C++, and Java names are case sensitive
  - The names in other languages are not
- Special words
  - An aid to readability; used to delimit or separate statement clauses
  - Def: A keyword is a word that is special only in certain contexts
    - i.e. in Fortran:
      - Real VarName (*Real is data type followed with a name, therefore Real is a keyword)*
      - Real = 3.4 (*Real is a variable)*
  - Disadvantage: poor readability
  - Def: A reserved word is a special word that cannot be used as a user-defined name

Variables – CO2

- A variable is an abstraction of a memory cell
- Variables can be characterized as a sextuple of attributes:
  (name, address, value, type, lifetime, and scope)
- Name - not all variables have them (anonymous)
- Address - the memory address with which it is associated (also called *l*-value)
  - A variable may have different addresses at different times during execution
  - A variable may have different addresses at different places in a program
  - If two variable names can be used to access the same memory location, they are called aliases
  - Aliases are harmful to readability (program readers must remember all of them)
- How aliases can be created:
  - Pointers, reference variables, C and C++ unions
  - Some of the original justifications for aliases are no longer valid; **e.g.**, memory reuse in FORTRAN
  - Replace them with dynamic allocation

- Type - determines the range of values of variables and the set of operations that are defined for values of that type; in the case of floating point, type also determines the Precision
- Value - the contents of the location with which the variable is associated
- Abstract memory cell - the physical cell or collection of cells associated with a variable

The Concept of Binding

- The *l*-value of a variable is its address
- The *r*-value of a variable is its value
- Def: A binding is an association, such as between an attribute and an entity, or between an operation and a symbol
- Def: Binding time is the time at which a binding takes place.
- Possible binding times:
  - Language design time--**e.g.**, bind operator symbols to operations
  - Language implementation time--**e.g.**, bind floating point type to a representation
  - Compile time--**e.g.**, bind a variable to a type in C or Java
  - Load time--**e.g.**, bind a FORTRAN 77 variable to a memory cell (or a C **static** variable)
  - Runtime--**e.g.**, bind a nonstatic local variable to a memory cell
- Def: A binding is static if it first occurs before run time and remains unchanged throughout program execution.
- Def: A binding is dynamic if it first occurs during execution or can change during execution of the program.
- Type Bindings
  - How is a type specified?
  - When does the binding take place?
  - If static, the type may be specified by either an explicit or an implicit declaration
- Def: An explicit declaration is a program statement used for declaring the types of variables
- Def: An implicit declaration is a default mechanism for specifying types of variables (the first appearance of the variable in the program)
- FORTRAN, PL/I, BASIC, and Perl provide implicit declarations
  - Advantage: writability
  - Disadvantage: reliability (less trouble with Perl)
- Dynamic Type Binding (JavaScript and PHP)
- Specified through an assignment statement **e.g.**, JavaScript

  list = [2, 4.33, 6, 8];

  list = 17.3;
  - Advantage: flexibility (generic program units)
  - Disadvantages:
    - High cost (dynamic type checking and interpretation)
    - Type error detection by the compiler is difficult
- Type Inferencing (ML, Miranda, and Haskell)
  - Rather than by assignment statement, types are determined from the context of the reference
- Storage Bindings & Lifetime
  - Allocation - getting a cell from some pool of available cells
  - Deallocation - putting a cell back into the pool

- Def: The lifetime of a variable is the time during which it is bound to a particular memory cell
- Categories of variables by lifetimes
  - Static--bound to memory cells before execution begins and remains bound to the same memory cell throughout execution.
    e.g., all FORTRAN 77 variables, C static variables
  - Advantages: efficiency (direct addressing), history-sensitive subprogram support
  - Disadvantage: lack of flexibility (no recursion)
- Categories of variables by lifetimes
  - Stack-dynamic--Storage bindings are created for variables when their declaration statements are elaborated.
  - If scalar, all attributes except address are statically bound
    e.g., local variables in C subprograms and Java methods
  - Advantage: allows recursion; conserves storage
  - Disadvantages:
    - Overhead of allocation and deallocation
    - Subprograms cannot be history sensitive
    - Inefficient references (indirect addressing)
- Categories of variables by lifetimes
  - Explicit heap-dynamic--Allocated and deallocated by explicit directives, specified by the programmer, which take effect during execution
  - Referenced only through pointers or references
    e.g., dynamic objects in C++ (via new and delete) all objects in Java
  - Advantage: provides for dynamic storage management
  - Disadvantage: inefficient and unreliable
- Categories of variables by lifetimes
  - Implicit heap dynamic--Allocation and deallocation caused by assignment statements
    e.g., all variables in APL; all strings and arrays in Perl and JavaScript
  - Advantage: flexibility
  - Disadvantages:
    - Inefficient, because all attributes are dynamic
    - Loss of error detection

2.5 Type Checking - CO3

- 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 as 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
- Def: A programming language is strongly typed if type errors are always detected

Type Compatibility

- Our concern is primarily for structured types
- Def: Name type compatibility means the two variables have compatible 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 compatible with integer types
  - Formal parameters must be the same type as their corresponding actual parameters (Pascal)
- Structure type compatibility means that two variables have compatible types if their types have identical structures
- More flexible, but harder to implement
- Consider the problem of two structured types:
- Are two record types compatible if they are structurally the same but use different field names?
- Are two array types compatible if they are the same except that the subscripts are different?
  (**e.g.**, [1..10] and [0..9])
- Are two enumeration types compatible if their components are spelled differently?
- With structural type compatibility, you cannot differentiate between types of the same structure (**e.g.**, different units of speed, both float)
- Language examples:
  - Pascal: usually structure, but in some cases name is used (formal parameters)
  - C: structure, except for records
  - Ada: restricted form of name
    - Derived types allow types with the same structure to be different
    - Anonymous types are all unique, even in:
      A, B : array (1..10) of INTEGER:

2.6 Strong Typing – CO2

- Advantage of strong typing: allows the detection of the misuses of variables that result in type errors
- Language examples:
  - FORTRAN 77 is not: parameters, *EQUIVALENCE*
  - Pascal is not: variant records
  - C and C++ are not: parameter type checking can be avoided; unions are not type checked
  - Ada is, almost (*UNCHECKED CONVERSION* is loophole)
    (Java is similar)
- 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

Named Constants

- Def: A named constant is a variable that is bound to a value only when it is bound to storage
- Advantages: readability and modifiability
- Used to parameterize programs
- The binding of values to named constants can be either static (called manifest constants) or dynamic
- Languages:
  - Pascal: literals only
  - FORTRAN 90: constant-valued expressions
  - Ada, C++, and Java: expressions of any kind

Variable Initialization

- Def: The binding of a variable to a value at the time it is bound to storage is called initialization
- Initialization is often done on the declaration statement **e.g.**, Java **int sum = 0**

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
- Case sensitivity and the relationship of names to special words represent design issues of names
- Variables are characterized by the sextuples: name, address, value, type, lifetime, scope
- Binding is the association of attributes with program entities
- Scalar variables are categorized as: static, stack dynamic, explicit heap dynamic, implicit heap dynamic
- Strong typing means detecting all type errors

Expressions and Statements & Control Structures

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.
- Arithmetic evaluation was one of the motivations for the development of the first programming languages.

2.7 Arithmetic Expressions – CO2, CO3

Arithmetic Expressions consist of operators, operands, parentheses and function calls.

Design Issues

Design issues for arithmetic expressions

- Operator precedence rules?
- Operator associativity rules?
- Order of operand evaluation?
- Operand evaluation side effects?
- Operator overloading?
- Type mixing in expressions?

Operators

- A **unary** operator has one operand.
- A **binary** operator has two operands.
- A **ternary** operator has three operands.

Operator Precedence Rules

The *operator precedence rules* for expression evaluation define the order in which adjacent operators of different precedence levels are evaluated.

Typical precedence levels:

- parentheses
- unary operators
- ** (if the language supports it)
- *, /
- +, -

Operator Associativity Rule

The *operator associativity rules* for expression evaluation define the order in which adjacent operators with the same precedence level are evaluated.

Typical associativity rules:

- Left to right, except **(Ruby and Fortran), which is right to left
- Sometimes unary operators associate right to left (**e.g.**, in FORTRAN)
  - APL is different; all operators have equal precedence and all operators associate right to left.
  - Precedence and associativity rules can be overridden with parentheses.

Conditional Expressions

Conditional Expressions (ternary operator ?:) available in C-based languages.
An example: (C, C++)

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

Evaluates as if written like

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

Operand Evaluation Order

Operand evaluation order as follows.

- Variables: fetch the value from memory.
- Constants: sometimes a fetch from memory; sometimes the constant is in the machine language instruction.
- Parenthesized expressions*:* evaluate all operands and operators first.
- The most interesting case is when an operand is a function call.

Potentials for Side Effects *Functional side effects:* when a function changes a two-way parameter or a non-local variable

Problem with functional side effects: When a function referenced in an expression alters another operand of the expression;

e.g., for a parameter change:

```
a = 10;
/* assume that fun changes its parameter */
b = a + fun(a);
```

Two possible solutions to the functional side effects problem: Write the language definition to disallow functional side effects.

- No two-way parameters in functions
- No non-local references in functions

Advantage: it works!

Disadvantages: inflexibility of one-way parameters and lack of non-local references

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

Disadvantage: limits some compiler optimizations

- Java requires that operands appear to be evaluated in left-to-right order.

Overloaded Operators

Use of an operator for more than one purpose is called *operator overloading.*

- Some are common (**e.g.**, + for int and float)
- Some are potential trouble (**e.g.**, * in C and C++)

Problems:

- Loss of compiler error detection (omission of an operand should be a detectable error)
- Some loss of readability
- Can be avoided by introduction of new symbols (**e.g.**, Pascal's **div** for integer division)
- C++, Ada, Fortran 95, and C# allow user-defined overloaded operators

Potential problems:

- Users can define nonsense operations.
- Readability may suffer, even when the operators make sense.

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.**, float to int*

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*

Mixed Mode

A *mixed-mode expression* is one that has operands of different types

A *coercion* is an implicit type conversion

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.
- In Ada, there are virtually no coercions in expressions

Explicit Type Conversions called as *casting* in C-based languages.

Examples:-

C: (int)angle, Ada: Float (Sum)

- *Note that Ada's syntax is similar to that of function calls*

Errors in Expressions causes

- Inherent limitations of arithmetic **e.g.**, division by zero
- Limitations of computer arithmetic **e.g.**, overflow
- Often ignored by the run-time system

Relational and Boolean Expressions

Relational Expressions:

- Use relational operators and operands of various types
- Evaluate to some Boolean representation
- Operator symbols used vary somewhat among languages (!=, /=, .NE., <>, #)
- JavaScript and PHP have two additional relational operator, === and !==
- Similar to their cousins, == and !=, except that they do not coerce their operands

Boolean Expressions:

Operands are Boolean and the result is Boolean

Example operators

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

- *No Boolean Type in C*
- C89 has 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:
- Left operator is evaluated, producing 0 or 1
- The evaluation result is then compared with the third operand (i.e., c)

2.8 Short Circuit Evaluation - CO3

An expression in which the result is determined without evaluating all of the operands and/or operators

Example: *(13*a) * (b/13–1)*

If 'a' is zero, there is no need to evaluate (b/13-1)

Problem with non-short-circuit evaluation

```
index = 1;
while (index <= length) && (LIST[index] != value)
index++;
```

- When index=length, LIST [index] will cause an indexing problem (assuming LIST has length -1 elements)
- 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 **|**)
- Ada: programmer can specify either (short-circuit is specified with and then and or else)
- Short-circuit evaluation exposes the potential problem of side effects in expressions **e.g.**, *(a > b)* **||** *(b++ / 3)*

2.9 Assignment Statements CO3

The general syntax: <target_var> <assign_operator> <expression>
The assignment operator

- '=' in FORTRAN, BASIC, the C-based languages
- ':=' in ALGOL, Pascal, Ada

–Equal '=' can be bad when it is overloaded for the relational operator for equality (that's why the C-based languages use == as the relational operator)
Conditional Targets (Perl)

```
($flag ? $total : $subtotal) = 0
```

Which is equivalent to

```
if ($flag){$total = 0}
else {$subtotal = 0}
```

Compound Assignment Operators

- A shorthand method of specifying a commonly needed form of assignment.
- Introduced in ALGOL; adopted by C
  Ex: *'a = a + b'* is written as *'a += b'*

Unary Assignment Operators

Unary assignment operators in C-based languages combine increment and decrement operations with assignment. For example
sum + = ++count (count incremented, added to sum)
sum + = count++ (count incremented, added to sum)
count++ (count incremented)
-count++ (count incremented then negated)

Assignment as an Expression

In C, C++, and Java, the assignment statement produces a result and can be used as operands.

```
while ((ch = getchar())!= EOF){...}
```

ch = getchar() is carried out; the result (assigned to ch) is used as a conditional value for the while statement

List Assignments

Perl and Ruby support list assignments

e.g., ($first, $second, $third) = (20, 30, 40);

Mixed-Mode Assignment

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.

2.10 Control Structures - CO3

A *control structure* is a control statement and the statements whose execution it controls.

Levels of Control Flow

- Within expressions
- Among program units
- Among program statements

Control Statements: Evolution

- FORTRAN I control statements were based directly on IBM 704 hardware
- Much research and argument in the 1960s about the issue

One important result: It was proven that all algorithms represented by flowcharts can be coded with only two-way selection and pretest logical loops

2.11 Selection Statements – CO3

A *selection statement* provides the means of choosing between two or more paths of execution. Two general categories:

- Two-way selectors
- Multiple-way selectors

Two-Way Selection Statements

General form as follows…

if control_expression
then clause
else clause

Design Issues:

- What is the form and type of the control expression?
- How are the **then** and **else** clauses specified?
- How should the meaning of nested selectors be specified?

The Control Expression

- If the 'then' reserved word or some other syntactic marker is not used to introduce the 'then' clause, the control expression is placed in parentheses.
- In C89, C99, Python, and C++, the control expression can be arithmetic.
- In languages such as Ada, Java, Ruby, and C#, the control expression must be Boolean.

Clause Form

- In many contemporary languages, the then and else clauses can be single statements or compound statements
- In Perl, all clauses must be delimited by braces (they must be compound)
- In Fortran 95, Ada, and Ruby, clauses are statement sequences
- Python uses indentation to define clauses

```
if x > y :
x = y
print "case 1"
```

*Nesting Selectors:*Java example

```
if (sum == 0)
if (count == 0)
result = 0;
else result = 1;
```

- Which if gets the else?
- Java's static semantics rule: else matches with the nearest if Nesting Selectors
- To force an alternative semantics, compound statements may be used:

```
if (sum == 0) {
if (count == 0)
result = 0;}
else result = 1;
```

- The above solution is used in C, C++, and C#
- Perl requires that all then and else clauses to be compound
- Statement sequences as clauses: **Ruby**

```
if sum == 0 then
if count == 0 then
result = 0
else
result = 1
end
end
```

-Python

```
if sum == 0 :
if count == 0 :
result = 0
else :
result = 1
```

Multiple-Way Selection Statements

Allow the selection of one of any number of statements or statement groups

Design Issues:

- What is the form and type of the control expression?
- How are the selectable segments specified?
- Is execution flow through the structure restricted to include just a single selectable segment?
- How are case values specified?
- What is done about unrepresented expression values?

Multiple-Way Selection: Examples

- *C, C++, and Java*

```
switch (expression) {
case const_expr_1: stmt_1;
…
case const_expr_n: stmt_n;
[default: stmt_n+1]}
```

- Design choices for C's **switch** statement
- Control expression can be only an integer type
- Selectable segments can be statement sequences, blocks, or compound statements
- Any number of segments can be executed in one execution of the construct (there is no implicit branch at the end of selectable segments)

- **default** clause is for unrepresented values (if there is no **default**, the whole statement does nothing)
- C#
  - Differs from C in that it has a static semantics rule that disallows the implicit execution of more than one segment
  - Each selectable segment must end with an unconditional branch (goto or break)
- Ada

```
case expression is
when choice list => stmt_sequence;
...
when choice list => stmt_sequence;
when others => stmt_sequence;]
end case;
```

More reliable than C's switch (once a stmt_sequence execution is completed, control is passed to the first statement after the case statement

- Ada design choices:
  1. Expression can be any ordinal type
  2. Segments can be single or compound
  3. Only one segment can be executed per execution of the construct
  4. Unrepresented values are not allowed
- Constant List Forms:
  1. A list of constants
  2. Can include:

- Subranges
- Boolean OR operators (|)

Multiple-Way Selection Using if

Multiple Selectors can appear as direct extensions to two-way selectors, using else-if clauses, for example in Python:

```
if count < 10 :
        bag1 = True
elsif count < 100 :
        bag2 = True
elseif count < 1000 :
        bag3 = True
```

Iterative Statements

- The repeated execution of a statement or compound statement is accomplished either by iteration or recursion
- General design issues for iteration control statements:
  1. How is iteration controlled?
  2. Where is the control mechanism in the loop?

Counter-Controlled Loops

A counting iterative statement has a loop variable, and a means of specifying the *initial* and *terminal*, and *stepsize* values

Design Issues:

- What are the type and scope of the loop variable?
- What is the value of the loop variable at loop termination?
- Should it be legal for the loop variable or loop parameters to be changed in the loop body, and if so, does the change affect loop control?
- Should the loop parameters be evaluated only once, or once for every iteration?

Iterative Statements: Examples

FORTRAN 95 syntax

DO *label var = start, finish [, stepsize]*

Stepsize can be any value but zero

Parameters can be expressions

Design choices:

1. Loop variable must be **INTEGER**
2. Loop variable always has its last value
3. The loop variable cannot be changed in the loop, but the parameters can; because they are evaluated only once, it does not affect loop control
4. Loop parameters are evaluated only once

- FORTRAN 95 : a second form:

  [name:] Do variable = initial, terminal [,stepsize]

  ...

  End Do [name]

  - Cannot branch into either of Fortran's Do statements
- Ada

  for var in [reverse] discrete_range loop ...

  end loop
- Design choices:
  - Type of the loop variable is that of the discrete range (A discrete range is a sub-range of an integer or enumeration type).
  - Loop variable does not exist outside the loop
  - The loop variable cannot be changed in the loop, but the discrete range can; it does not affect loop control
  - The discrete range is evaluated just once
  - Cannot branch into the loop body
- C-based languages

  for ([expr_1] ; [expr_2] ; [expr_3]) statement
  - The expressions can be whole statements, or even statement sequences, with the statements separated by commas
  - The value of a multiple-statement expression is the value of the last statement in the expression
  - If the second expression is absent, it is an infinite loop
- Design choices:
  - There is no explicit loop variable
  - Everything can be changed in the loop
  - The first expression is evaluated once, but the other two are evaluated with each iteration
- C++ differs from C in two ways:
  - The control expression can also be Boolean
  - The initial expression can include variable definitions (scope is from the definition to the end of the loop body)
- Java and C#
  - Differs from C++ in that the control expression must be Boolean
  - Iterative Statements: Logically-Controlled Loops
  - Repetition control is based on a Boolean expression
- Design issues:
  - Pretest or posttest?
  - Should the logically controlled loop be a special case of the counting loop statement or a separate statement?

- *Iterative Statements: Logically-Controlled Loops: Examples*
  C and C++ have both pretest and posttest forms, in which the control expression can be arithmetic:
  ```
  while (ctrl_expr) do
  loop body loop body
  while (ctrl_expr)
  ```
- Java is like C and C++, except the control expression must be Boolean (and the body can only be entered at the beginning -- Java has no **goto**
- Iterative Statements: Logically-Controlled Loops: Examples
- Ada has a pretest version, but no posttest
- FORTRAN 95 has neither
- Perl and Ruby have two pretest logical loops, while and until. Perl also has two posttest loops

Unconditional Branching: *User-Located Loop Control Mechanisms*

- Sometimes it is convenient for the programmers to decide a location for loop control (other than top or bottom of the loop)
- Simple design for single loops (**e.g.**, break)
- Design issues for nested loops
  - Should the conditional be part of the exit?
  - Should control be transferable out of more than one loop?

User-Located Loop Control Mechanisms break and continue

- C , C++, Python, Ruby, and C# have unconditional unlabeled exits (**break)**
- Java and Perl have unconditional labeled exits (**break** in Java, **last** in Perl)
- C, C++, and Python have an unlabeled control statement, **continue**, that skips the remainder of the current iteration, but does not exit the loop
- Java and Perl have labeled versions of **continue**

Iterative Statements: Iteration Based on Data Structures

- Number of elements of in a data structure control loop iteration
- Control mechanism is a call to an *iterator* function that returns the next element in some chosen order, if there is one; else loop is terminate
- C's **for** can be used to build a user-defined iterator:
  ```
  for (p=root; p==NULL;
  traverse(p)){ }
  ```
- C#'s **foreach** statement iterates on the elements of arrays and other collections:
  ```
  Strings[] = strList = {"Bob", "Carol", "Ted"};
  foreach (Strings name in strList)
  Console.WriteLine ("Name: {0}", name);
  ```
  The notation {0} indicates the position in the string to be displayed
- Perl has a built-in iterator for arrays and hashes, **foreach** Unconditional Branching
- Transfers execution control to a specified place in the program
- Represented one of the most heated debates in 1960's and 1970's
- Well-known mechanism: goto statement
- Major concern: Readability
- Some languages do not support goto statement (**e.g.**, Java)
- C# offers goto statement (can be used in switch statements)
- Loop exit statements are restricted and somewhat camouflaged goto's