UNIT 2

Names and its Design issues.

Imperative languages are abstractions of von Neumann architecture

- A machine consists of
  - Memory - stores both data and instructions
  - Processor - can modify the contents of memory
- Variables are used as an abstraction for memory cells
  - For primitive types correspondence is direct
  - For structured types (objects, arrays) things are more complicated

Names

- Why do we need names?
  - need a way to refer to variables, functions, user-defined types, labeled statements,
    Design issues for names:
    - Maximum length?
    - What characters are allowed?
    - Are names case sensitive?
      - C, C++, and Java names are case sensitive
      - this is not true of other languages
    - Are special words reserved words or keywords?

Length of Names

- 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 implementers often impose one

Keywords vs Reserved Words

- Words that have a special meaning in the language
- A *keyword* is a word that is special only in certain contexts, e.g., in Fortran

  \endash Real VarName (Real *is a data type followed with a name, therefore*

  Real *is a keyword)*

  \endash Real = 3.4 (*Real is a variable)*

- A *reserved word* is a special word that cannot be used as a user-defined name
  - most reserved words are also keywords

Variables

- A variable is an abstraction of a memory cell
- Variables can be characterized as a sextuple of attributes:
  - Name
  - Address
  - Value
  - Type
  - Lifetime
  - Scope

Variable Attributes

- Name - not all variables have them
- Address - the memory address with which it is associated (l-value)
- *Type* - allowed range of values of variables and the set of defined operations
- *Value* - the contents of the location with which the variable is associated (r-value)

The concept of Binding and possible

Binding times. The Concept of Binding

- A *binding* is an association, such as between an attribute and an entity, or between an operation and a symbol

- entity could be a variable or a function or even a class

- *Binding time* is the time at which a binding takes place.

Possible Binding Times

- Language design time -- bind operator symbols to operations
- Language implementation time-- bind floating point type to a representation
- Compile time -- bind a variable to a type in C or Java
- Load time -- bind a FORTRAN 77 variable to a memory cell (or a C static variable)
- Runtime -- bind a nonstatic local variable to a memory cell

Static and Dynamic Binding

- A binding is *static* if it first occurs before run time and remains unchanged throughout program execution.
- A binding is *dynamic* if it first occurs during execution or can change during execution of the program

Type Binding

\endash How is a type specified?

- An *explicit declaration* is a program statement used for declaring the types of variables
- An *implicit declaration* is a default mechanism for specifying types of variables (the first appearance of the variable in the program)

\endash When does the binding take place?

\endash If static, the type may be specified by either an explicit or an implicit declaration

Type checking and how coercion related to type

checking? Type Checking

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

- A *type error* is the application of an operator to an operand of an inappropriate type

When is type checking done?

- If all type bindings are static, nearly all type checking can be static (done at compile time)
- If type bindings are dynamic, type checking must be dynamic (done at run time)
- A programming language is *strongly typed* if type errors are always detected
  - Advantage: allows the detection of misuse of variables that result in type errors

How strongly typed?

  - FORTRAN 77 is not: parameters, EQUIVALENCE
  - Pascal has variant records
  - C and C++
    - parameter type checking can be avoided
    - unions are not type checked
  - Ada is almost
    - UNCHECKED CONVERSION is loophole
  - Java is similar

Coercion

- The automatic conversion between types is called coercion.
- Coercion rules they can weaken typing considerably
  - C and C++ allow both widening and narrowing coercions
  - Java allows only widening coercions
- Java's strong typing is still far less effective than that of Ada

Dynamic type binding

- Dynamic Type Binding (Perl, JavaScript and PHP, Scheme)
- Specified through an assignment statement e.g., JavaScript

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

list = 17.3;
```

- This provides a lot of flexibility

Storage Bindings & Lifetime

- The lifetime of a variable is the time during which it is bound to a particular memory cell
  - Allocation - getting a cell from some pool of available cells
  - Deallocation - putting a cell back into the pool
- Depending on the language, allocation can be either controlled by the programmer or done automatically

Categories of Variables by Lifetimes

- Static--lifetime is same as that of the program
- Stack-dynamic--lifetime is duration of subprogram
- *Explicit heap-dynamic* -- Allocated and deallocated by explicit directives, specified by the programmer, which take effect during execution
- *Implicit heap-dynamic*--Allocation and deallocation caused by assignment statements

Variable Scope

- The *scope* of a variable is the range of statements over which it is visible
- The *nonlocal variables* of a program unit are those that are visible but not declared there
- The scope rules of a language determine how references to names are associated with variables
- Scope and lifetime are sometimes closely related, but are different concepts

Static Scope

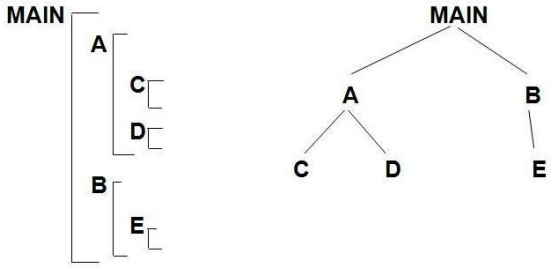
- The scope of a variable can be determined from the program text
- To connect a name reference to a variable, you (or the compiler) must find the declaration
- Search process: search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name
- Enclosing static scopes (to a specific scope) are called its static ancestors; the nearest static ancestor is called a static parent

Scope and Shadowed Variables

- Variables can be hidden from a unit by having a "closer" variable with the same name
- C++, Java and Ada allow access to some of these "hidden" variables
  - In Ada: **unit.name**
  - In C++: **class_name::name** or **::name** for globals
  - In Java: **this.name** for variables declared at class level

Static Scoping and Nested Functions

- Assume MAIN calls A and B
  A calls C and D
  B calls A and E



Nested Functions and Maintainability

- Suppose the spec is changed so that D must now access some data in B
- Solutions:
  - Put D in B (but then C can no longer call it and D cannot access A's variables)
  - Move the data from B that D needs to MAIN (but then all procedures can access them)
- Same problem for procedure access

- Using nested functions with static scoping often encourages many globals
  - not considered to be good programming practice
  - Current thinking is that we can accomplish the same thing better with modules (classes)
- Static scoping is still preferred to the alternative
  - Most current languages use static scoping at the block level even if they don't allow nested functions

Storage Bindings & Lifetime:

The lifetime of a variable is the time during which it is bound to a particular memory cell

- **Allocation -** getting a cell from some pool of available cells
- **Deallocation -** putting a cell back into the pool

– Depending on the language, allocation can be either controlled by the programmer or done automatically

Categories of Variables by Lifetimes:

- **Static--**lifetime is same as that of the program
- **Stack-dynamic**--lifetime is duration of subprogram
- ***Explicit heap-dynamic*** -- Allocated and deallocated by explicit directives, specified by the programmer, which take effect during execution***Implicit heap-dynamic***--Allocation and deallocation caused by assignment statements

Variable Scope:

- The ***scope* of a variable** is the range of statements over which it is **visible**
- The ***nonlocal variables*** of a program unit are those that are **visible** but **not declared** there

– The scope rules of a language determine how references to names are associated with variables

– Scope and lifetime are sometimes closely related, but are different concepts

Static Scope:

– The **scope of a variable** can be determined from the **program text**

– To connect a name reference to a variable, you (or the compiler) must find the declaration

– **Search process**: search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name **Enclosing static scopes** (to a specific scope) are **called its static ancestors**; the nearest static ancestor is called a static parent

Scope and Shadowed Variables:

– Variables can be hidden from a unit by having a "**closer**" variable with the same name

– **C++, Java and Ada** allow access to some of these "hidden" variables

  – **In Ada**: **unit.name**

  – **In C++: class_name::name** or **::name** for globals

  – **In Java**: **this.name** for variables declared at class level

Static Scoping and Nested Functions:

- Assume MAIN calls A and B A calls C and D B calls A and E

Nested Functions and Maintainability:

- Suppose the spec is changed so that D must now access some data in B
- **Solutions:**
  - Put D in B (but then C can no longer call it and D cannot access A's variables)
  - Move the data from B that D needs to MAIN (but then all procedures can access them)
- Same problem for procedure access

Dynamic Scope:

- Based on calling sequences of program units, not their textual layout (temporal versus spatial)
- References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point
- This is easy to implement but it makes programs hard to follow
- Used in APL, SNOBOL, early versions of LISP

- Perl allows you to use dynamic scope for selected variablesScope

Static scoping

- Reference to x is to MAIN's x

Dynamic scoping

- Reference to x is to SUB1's x

Evaluation of Dynamic Scoping:

- **Advantage**: convenience

- **Disadvantage:** poor readability

Referencing Environments:

– The *referencing environment* of a statement is the collection of all names that are visible in the statement

– In a **static-scoped language**, it is the local variables plus all of the visible variables in all of the enclosing scopes

– In a **dynamic-scoped language**, the referencing environment is the local variables plus all visible variables in all active subprograms subprogram is active if its execution has begun but has not yet terminate

–

DATA TYPES

Data type Definition:

- Collection of data objects
- a set of predefined operations

- **descriptor** : collection of attributes for a variable
- **object** :instance of a user-defined (abstract data) type

Data Types:

- **Primitive**
  - not defined in terms of other data types
  - defined in the language
  - often reflect the hardware
- **Structured**
  - built out of other types

Integer Types:

Usually based on hardware

May have several ranges

- **Java's signed integer sizes**: byte, short, int, long
- **C/C++** have unsigned versions of the same types
- **Scripting languages** often just have one integer type
- **Python** has an integer type and a long integer which can get as big as it needs to.

- **Representing Integers:**
    - Can convert positive integers to base
    - How do you handle negative numbers with only 0s and 1s?

- Sign bit
- Ones complement
- Twos complement - this is the one that is used
- Representing negative integers.

Representing negative integers:

- Sign bit
- Ones complement

Twos Complement:

To get the binary representation, take the complement and add 1

Floating Point Types:

Model real numbers only an approximation due to round-off error

- For scientific use support at least two floating-point types (e.g., float and double; sometimes more)
- 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
- Floating-point values are represented as fractions and exponents, a form that is borrowed from scientific notation

– 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.

– Usually based on hardware

– IEEE Floating-Point Standard 754

–32 and 64 bit standards

Representing Real Numbers:

– We can convert the decimal number to base 2 just as we did for integers

– How do we represent the decimal point?

– fixed number of bits for the whole and fractional parts severely limits the range of values we can represent

– Use a representation similar to scientific notation

IEEE Floating Point Representation:

- Normalize the number
    - one bit before decimal point
- Use one bit to represent the sign (1 for negative)
- Use a fixed number of bits for the exponent which is offset to allow for negative exponents

Floating Point Types:

- C, C++ and Java have two floating point types
    - float
    - double
- Most scripting languages have one floating point type
    - Python's floating point type is equivalent to a C double
- Some scripting languages only have one kind of number which is a floating point type

Fixed Point Types (Decimal) :

- For business applications (money) round-off errors are not acceptable
    - Essential to COBOL

- .NET languages have a decimal data type

- store a fixed number of decimal digits

- Operations generally have to be defined in software

- ***Advantage***: accuracy

- ***Disadvantages***: limited range, wastes memory

C# decimal Type:

- 128-bit representation

-28 28

- **Range:** 1.0x10 to 7.9x10

- **Precision**: representation is exact to 28 or 29 decimal places (depending on size of number)

- no roundoff error

Other Primitive Data Types:

- **Boolean**

- Range of values: two elements, one for ―true‖ and one for ―false‖

- Could be implemented as bits, but often as bytes

– Boolean types are often used to represent switches or flags in programs.

– A Boolean value could be represented by a single bit, but because a single bit of memory cannot be accessed efficiently on many machines, they are often stored in the smallest efficiently addressable cell of memory, typically a byte.

- **Character**

  - Stored as numeric codings

  - Most commonly used coding: ASCII

  - An alternative, 16-bit coding: Unicode

- Complex (Fortran, Scheme, Python)

- ational (Scheme)

Character Strings :

- Values are sequences of characters

– Character string constants are used to label output, and the input and output of all kinds of data are often done in terms of strings.

- **Operations:**

  - Assignment and copying

- Comparison (=, >, etc.)
- Catenation
- Substring reference
- Pattern matching
- A **substring reference** is a reference to a substring of a given string. Substring references are discussed in the more general context of arrays, where the substring references are called **slices**.
- In general, both assignment and comparison operations on character strings are complicated by the possibility of string operands of different lengths.

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

Character String Implementations:

- **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
- **Java**
    - String class

String Length Options

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

String Implementation

- **Static length:** compile-time descriptor
- **Limited dynamic length**: may need run-time descriptor

- not in C and C++

- **Dynamic length:** needs run-time descriptor;

    - allocation/deal location is main implementation issue

User-Defined Ordinal Types:

- **Cordinal type** : range of possible values corresponds to set of positive integers

- **Primitive ordinal types**

    - integer

    - char

    - boolean

- **User-defined ordinal types**

    - enumeration types

    - ubrange types

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

– duplication of names

– coercion rules

Enums in C (and C++):

– To define an enumerated type in **C**

Ex:

Enum weekday {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday}; Enum weekday today = Tuesday;

– Use **typedef** to give the type a name

typedef enum weekday {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday} weekday;

Weekday today = Tuesday;

By default, values are consecutive starting from 0.

- You can explicitly assign values **Enum months {January=1, February,};**

Enumerations in Java 1.5:

- An enum is a new class which extends **java.lang.Enum** and implements Comparable
  - Get type safety and compile-time checking
  - Implicitly public, static and final
  - Can use either == or equals to compare
  - toString and valueOf are overridden to make input and output easier

Java enum Example:

- **Defining an enum type**

  Enum Season {WINTER, SPRING, SUMMER, FALL};

- **Declaring an enum variable**

Season season = Season. WINTER;

– to String gives you the **string representation of the name**

System.out.println (season);

prints WINTER

– **valueOf l**ets you convert a String to an enum Season = valueOf (—SPRING‖);

Sub range Types:

– A contiguous subsequence of an ordinal type

Example:

– **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;

-

Evaluation

Subrange types enhance readability by making it clear to readers that variables of subtypes can store only certain ranges of values. Reliability is increased with subrange types, because assigning a value to a subrange variable that is outside the specified range is detected as an error, either by the compiler (in the case of the assigned value being a literal value) or by the run-time system (in the

case of a variable or expression). It is odd that no contemporary language except Ada has subrange types.

Implementation of User-Defined Ordinal Types

– Enumeration types are implemented as integers

– Subrange types are implemented like the parent types

– code inserted (by the compiler) to restrict assignments to subrange variables

Arrays and Indices

– Specific elements of an array are referenced by means of a two-level syntactic mechanism, where the first part is the aggregate name, and the second part is a possibly dynamic selector consisting of one or more items known as **subscripts** or **indices**.

– If all of the subscripts in a reference are constants, the selector is static; otherwise, it is dynamic. The selection operation can be thought of as a mapping from the array name and the set of subscript values to an element in the aggregate. Indeed, arrays are sometimes called **finite mappings**. Symbolically, this mapping can be shown as

array_name(subscript_value_list) → element

Subscript Bindings and Array Categories

– The binding of the subscript type to an array variable is usually static, but the subscript value ranges are sometimes dynamically bound.

– There are five categories of arrays, based on the binding to subscript ranges, the binding to storage, and from where the storage is allocated.

– 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 i**s that the storage for the array is fixed for the entire execution time of the program.

– 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

The **disadvantage** is the required allocation and deallocation time

– 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

– 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 **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.

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:

– The **disadvantage** is that allocation and deallocation take longer and may happen many times during execution of the program.

Array Initialization

- Some languages provide the means to initialize arrays at the time their storage is allocated.
- An array aggregate for a single-dimensioned array is a list of literals delimited by parentheses and slashes. For example, we could have

```
Integer, Dimension (3) :: List = (/0,  5,5/)
```

In the C declaration

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

These arrays can be initialized to string constants,
as in **char** name [] = "freddie";

Arrays of strings in C and C++ can also be initialized with string literals. In this case, the array is one of pointers to characters.

For example,

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

In Java, similar syntax is used to define and initialize an array of references to String objects. For example,

String[] names = ["Bob", "Jake", "Darcie"];

Ada provides two mechanisms for initializing arrays in the declaration statement: by listing them in the order in which they are to be stored, or by directly assigning them to an index position using the => operator, which in Ada is called an **arrow**.

For example, consider the following:

List : **array** (1.5) **of** Integer := (1, 3, 5, 7, 9); Bunch :

array (1.5) **of** Integer := (1 => 17, 3 => 34,

others => 0);

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 of the columns have the same number of elements. Rectangular arrays model rectangular tables exactly.

A **jagged array** is one in which the lengths of the rows need not be the same.

For example, a jagged matrix may consist of three rows, one with 5 elements, one with 7 elements, and one with 12 elements. This also applies to the columns and higher dimensions. So, if there is a third dimension (layers), each layer can have a different number of elements. Jagged arrays are made possible when multi dimensioned arrays are actually arrays of arrays. For example, a matrix would appear as an array of single-dimensioned arrays.

For example,

myArray[3][7]

Slices:

- A **slice** of an array is some substructure of that array.

For example, if A is a matrix, then the first row of A is one possible slice, as are the last row and the first column. It is important to realize that a slice is not a new data type. Rather ,it is a mechanism for referencing part of an array as a unit.

Evaluation

Arrays have been included in virtually all programming languages

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. There is no way to pre compute the address to be accessed by a reference such as list [k]

A single-dimensioned array is implemented as a list of adjacent memory cells. Suppose the array list is defined to have a subscript range lower bound of 0. The access function for list is often of the form address (list [k]) = address (list [0]) + k * element_size where the first operand of the addition is the constant part of the access function, and the second is the variable part .

If the element type is statically bound and the array is statically bound to storage, then the value of the constant part can be computed before run time. However, the addition and multiplication operations must be done at run time.

The generalization of this access function for an arbitrary lower bound is address (list[k])
= address (list [lower_bound]) + ((k - lower_bound) * element_size)

Associative Arrays

An **associative array** is an unordered collection of data elements that are indexed by an equal number of values called **keys**. In the case of non-associative arrays, the indices never need to be stored (because of their regularity). In an associative array, however, the user-defined keys must be stored in the structure. So each element of an associative array is in fact a pair of entities, a key and a value. We use Perl‘s design of associative arrays to illustrate this data structure. Associative arrays are also supported directly by Python, Ruby, and Lua and by the standard class libraries of Java, C++, C#, and F#. The only design issue that is specific for associative arrays is the form of references to their elements.

Structure and Operations

In Perl, associative arrays are called **hashes**, because in the implementation their elements are stored and retrieved with hash functions. The namespace for Perl hashes is distinct: Every hash variable name must begin with a percent sign (%). 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); Recall that scalar variable names begin with dollar signs ($). For example,

$salaries {"Perry"} = 58850;

A new element is added using the same assignment statement form. An element can be removed from the hash with the **delete** operator, as in

Delete $salaries{"Gary"};

Record Types

A **record** is an aggregate of data elements in which the individual elements are identified by names and accessed through offsets from the beginning of the structure. There is frequently a need in programs to model a collection of data in which the individual elements are not of the same type or size. For example, information about a college student might include name, student

number, grade point average, and so forth. A data type for such a collection might use a character string for the name, an integer for the student number, a floating point for the grade point average, and so forth. Records are designed for this kind of need.

The following **design issues** are specific to records:

- What is the syntactic form of references to fields?
- Are elliptical references allowed?

Definitions of Records

The fundamental difference between a record and an array is that record elements ,or **fields**, are not referenced by indices. Instead, the fields are named with identifiers, and references to the fields are made using these identifiers The COBOL form of a record declaration, which is part of the data division of a COBOL program, is illustrated in the following example:

– EMPLOYEE-RECORD.

– EMPLOYEE-NAME.

FIRST PICTURE IS X(20).

MIDDLE PICTURE IS X(10).

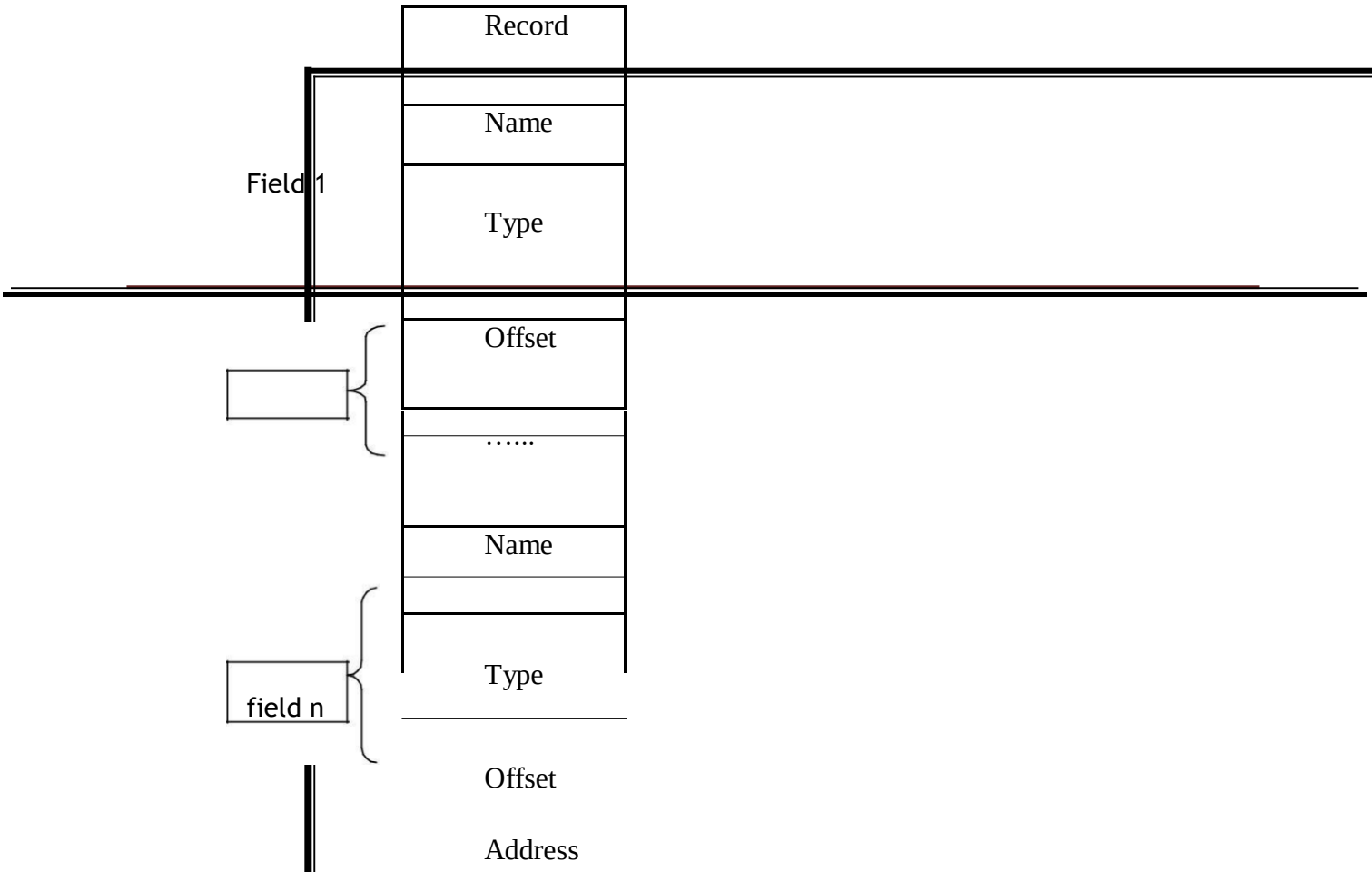
LAST PICTURE IS X(20).

02 HOURLY-RATE PICTURE IS 99V99.

The EMPLOYEE-RECORD record consists of the EMPLOYEE-NAME record and the HOURLY-RATE field. The numerals 01, 02, and 05 that begin the lines of the record declaration are **level numbers**, which indicate by their relative values the hierarchical structure of the record

Implementation of Record Types

The fields of records are stored in adjacent memory locations. But because the sizes of the fields are not necessarily the same, the access method used for arrays is not used for records. Instead, the offset address, relative to the beginning of the record, is associated with each field. Field accesses are all handled using these offsets. The compile-time descriptor for a record has the general form shown in Figure 6.7. Run-time descriptors for records are unnecessary



Union Types

A **union** is a type whose variables may store different type values at different times during program execution. As an example of the need for a union type, consider a table of constants for a compiler, which is used to store the constants found in a program being compiled. One field of each table entry is for the value of the constant. Suppose that for a particular language being compiled, the types of constants were integer, floating point, and Boolean. In terms of table management, it would

be convenient if the same location, a table field, could store a value of any of these three types. Then all constant values could be addressed in the same way. The type of such a location is, in a sense, the union of the three value types it can store.

Design Issues

The problem of type checking union types, leads to one major design issue. The other fundamental question is how to syntactically represent a union. In some designs, unions are confined to be parts of record structures, but in others they are not. So, the primary design issues that are particular to union types are the following:

Should type checking be required? Note that any such type checking must be dynamic.

Should unions be embedded in records?

Ada Union Types

The Ada design for discriminated unions, which is based on that of its predecessor language, Pascal, allows the user to specify variables of a variant record type that will store only one of the possible type values in the variant. In this way, the user can tell the system when the type checking can be static. Such a restricted variable is called a **constrained variant variable**.

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 record like structure. At compile time, the complete description of each variant must be stored. This can be done by associating a case table with the tag entry in the descriptor. The case table has an entry for each variant, which points to a descriptor for that particular variant. To illustrate this arrangement, consider the following Ada example:

type Node (Tag : Boolean) **is record**

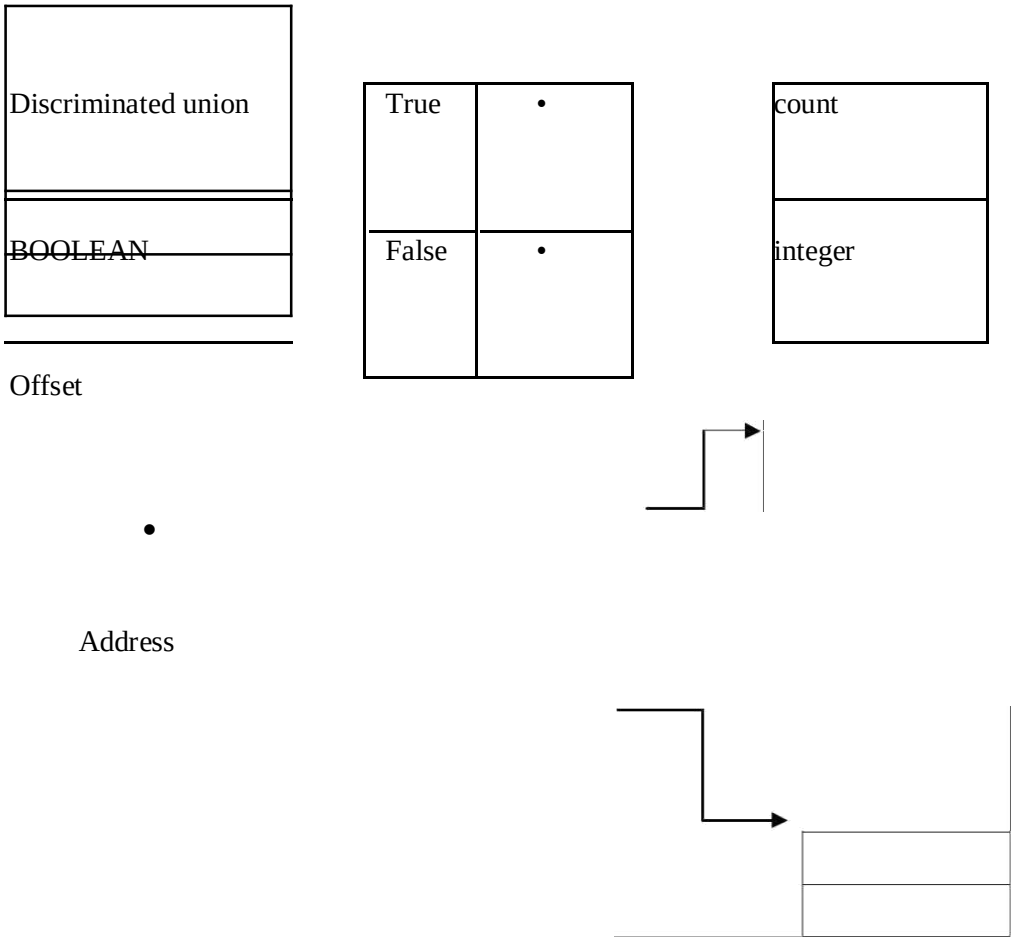
case Tag **is**

when True => Count : Integer;

when False => Sum : Float;

end case; **end record**;

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



Pointer and Reference Types

– A *pointer* is a variable whose value is an address

  – range of values that consists of memory addresses plus 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*)

– Generally represented as a single number

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

Pointers in C and C++:

– Extremely flexible but must be used with care

– Pointers can point at any variable regardless of when 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)

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 refer to call instances

– **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

Expressions and Statements Arithmetic Expressions:

– Arithmetic evaluation was one of the motivations for computers

– In programming languages, arithmetic expressions consist of operators, operands, parentheses, and function calls.

– An operator can be **unary**, meaning it has a single operand, **binary**, meaning it has two operands, or **ternary**, meaning it has three operands.

– In most programming languages, binary operators are **infix**, which means they appear between their operands.

– One exception is Perl, which has some operators that are **prefix**, which means they precede their operands.

– 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, usually from memory, and executing arithmetic operations on those operands.

– Arithmetic expressions consist of

- operators
- operands
- parentheses
- function calls

Issues for Arithmetic Expressions:

- operator precedence rules
- operator associatively rules
- order of operand evaluation
- operand evaluation side effects
- operator overloading
- mode mixing expressions

Operators:

A unary operator has one operand

- unary -, !

- A binary operator has two operands

- +,-,*,/,%

- A ternary operator has three operands - ?:

Conditional Expressions:

- a ternary operator in C-based languages (e.g., C, C++)

- An example:

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

- Evaluates as if written like

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

Operator Precedence Rules:

– *Precedence rules* 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 Associatively Rules:

– A*ssociativity rules* define the order in which adjacent operators with the same precedence level are evaluated

– Typical associativity rules

  – Left to right for arithmetic operators

  – exponentiation (** or ^) 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 overriden with parentheses

Operand Evaluation Order:

– 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

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);

Functional Side Effects:

– Two possible solutions to the 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!

Disadvantage: inflexibility of two-way parameters and non-local references

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

Disadvantage: limits some compiler optimizations

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++)

– 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)

Type Conversions

A ***narrowing conversion*** converts an object to a type that does not include all of the values of the original type

– e.g., float to int

– A ***widening conversion*** converts an object to a type that can include at least approximations to all of the values of the original type

– e.g., int to float

Coercion:

– 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, widening conversions are allowed to happen implicitly

– In Ada, there are virtually no coercions in expressions

Casting:

– Explicit Type Conversions

– Called *casting* in C-based language

– Examples

- **C:** (int) angle

- **Ada:** Float (sum)

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

Errors in Expressions:

Causes

– Inherent limitations of arithmetic e.g., division by zero, round-off errors

– Limitations of computer arithmetic e.g. overflow

– Often ignored by the run-time system

Relational Operators:

- Use operands of various types

- Evaluate to some Boolean representation

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

Boolean Operators:

- Operands are Boolean and the result is Boolean

- **Example o**perators

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

No Boolean Type in C:

- C has no Boolean type

- it uses int type with 0 for false and nonzero for true

Consequence

– a < b < c is a legal expression

– the result is not what you might expect:

Left operator is evaluated, producing 0 or 1

– This result is then compared with the third operand (i.e., c)

Precedence of C-based operators:

postfix ++, --

unary +, -, prefix ++, --, !

*,/,% binary +, -

<, >, <=, >=

=, !=

&&

||

Short Circuit Evaluation:

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)

– Usually used for logical operators

– Problem with non-short-circuit evaluation

While (index <= length) && (LIST[index] != value) Index++;

– When index=length, LIST [index] will cause an indexing problem (assuming LIST has length -1 elements)

Mixed-Mode Assignment:

Assignment statements can also be mixed-mode, for **example** int a, b; float c; c = a / b;

– In **Java**, only widening assignment coercions are done

– In **Ada,** there is no assignment coercion

Assignment Statements:

The general syntax

<target_var> <assign_operator> <expression>

The assignment operator

– **FORTRAN, BASIC, PL/I, C, C++, Java** := **ALGOLs, Pascal, Ada**

= can be bad when it is overloaded for the relational operator for equality

Compound Assignment:

– A shorthand method of specifying a commonly needed form of assignment

– Introduced in **ALGOL;** adopted by **C**

–Example 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

– These have side effects

– Examples

sum = ++count (count incremented, added to sum) sum = count++ (count added to sum, incremented) Count++ (count incremented) -count++ (count incremented then negated - right-associative)

Assignment as an Expression:

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

– **An example:**

While ((ch = get char ())!= EOF){…}

ch = get char() is carried out; the result (assigned to ch) is used in the condition for the while

statement

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

Control Structure

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

– Design question

–Should a control structure have multiple entries?

Selection Statements

– 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:

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 (continued)

– 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)

Multiple-Way Selection: Examples

– **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:

– Expression can be any ordinal type

– Segments can be single or compound

– Only one segment can be executed per execution of the construct

– Unrepresented values are not allowed

Constant List Forms:

– A list of constants

– 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

elif 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:

– How is iteration controlled?

– 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**:

- Loop variable must be **INTEGER**
- Loop variable always has its last value

- 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

- 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

Iterative Statements: 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?

Iterative Statements: 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

Guarded Commands

Designed by Dijkstra

Purpose: to support a new programming methodology that supported verification (correctness) during development

Basis for two linguistic mechanisms for concurrent programming (in CSP and Ada)

Basic Idea: if the order of evaluation is not important, the program should not specify one

Selection Guarded Command

•Form

if <Boolean exp> -> <statement> [] <Boolean exp> -> <statement>

...

[] <Boolean exp> -> <statement> fi

•Semantics: when construct is reached,

–Evaluate all Boolean expressions

–If more than one are true, choose one non-deterministically

–If none are true, it is a runtime error Selection Guarded Command: Illustrated

Loop Guarded Command

Form

do <Boolean> -> <statement>

[] <Boolean> -> <statement>

...

[] <Boolean> -> <statement> od

Semantics: for each iteration

Evaluate all Boolean expressions

If more than one are true, choose one non-deterministically; then start loop again

If none are true, exit loop

Guarded Commands: Rationale

Connection between control statements and program verification is intimate

Verification is impossible with goto statements

Verification is possible with only selection and logical pretest loops

Verification is relatively simple with only guarded commands