UNIT 2: DATA, DATA TYPES, AND BASIC STATEMENTS

SYLLABUS: Names, variables, binding, type checking, scope, scope rules, lifetime and garbage collection, primitive data types, strings, array types, associative arrays, record types, union types, pointers and references, Arithmetic expressions, overloaded operators, type conversions, relational and boolean expressions, assignment statements , mixed mode assignments, control structures – selection, iterations, branching, guarded Statements

2.1 NAMES: Names are also associated with labels , subprograms, formal parameters and other constructs. The term *identifier* is often used interchangeably with name.

2.1.1 DESIGN ISSUES: The following are primary design issues for names.

- Are names are case sensitive?
- Are special words, reserved words or keywords?

2.1.2 NAME FORMS: A **name** is a string of characters used to identify some entity in a program.

- Fortran 95 allows up to 31 characters in its names.
- C 89 had no length limitations in its internal names, but only the first 31 were significant.
- Names in java and ADA have no length limit, and all the characters in them are significant.
- Names is most programming languages have the same form: a letter followed by a string consisting of letters , digits, and underscore characters(_).
- The use of underscore characters to form names was widely used in the 1970's and 1980's.
- In the newer versions of Fortran prior and Fortran 90, names could have embedded spaces, which were ignored. For example, the following two names were equivalent:
  - Sum of salaries
  - Sumofsalaries
- The C-based languages, uppercase and lowercase letters in names are distinct; that is, names in these languages are **case sensitive.** For example , the following three names are distinct in C++: rose, ROSE, and Rose.
- But in the languages whose variables names are case sensitive, although Rose and rose look similar.

- In versions of Fortran prior to 90, only uppercase letters could be used in names.
- Fortran 77 allow lowercase letters ; they are simply translate them to uppercase letters for internal use during compilation.

2.1.3 SPECIAL WORDS: Special words in programming languages are used to make programs more readable by naming actions to be performed.

- These words are classified as reserved words, but in some they are only keywords.
  - ➢ **KEYWORD:** A **keyword** is a word of a programming language that is special only in certain contexts. Fortran is one of the languages whose special words are keyword. Fortran, the word Real, when found at the beginning of a statement and followed by a name, is a declarative statement. If the word Real is followed by the assignment operator, it is considered a variable name. These two use are illustrated in the following:
    - ▪ Real Apple
    - ▪ Real= 3.4
  - ➢ **RESERVED WORD:** A **reserved word** is a special word of a programming language that cannot be used as a name. As a language design choice, reserved words are better than keywords because the ability to redefine keywords can lead to readability problems. For example, in Fortran, one could have the statements:
    - ▪ Integer Real
    - ▪ Real Integer
      - Which declare the program variable Real to be of Integer type and variable able Integer to be of Real type. In addition to the strange appearance of these declaration statements, the appearance of Real and Integer as variable names elsewhere in the program could be misleading to program readers.

2.2 VARIABLES: A program variable is an abstraction of a computer memory cell or collection of cells. Programmers often think of variables as names for memory locations, but there is much more to a variable than just a name.

➢ The translator that converted the names to actual addresses also chose those addresses.

- A variable can be characterized as a sextuple of attributes :(names, address, value, type, lifetime, scope).
- It provides the clearest way to explain the various aspects of variables.
- The variable attributes will lead to examinations of the important related concepts of aliases, binding, binding times, declarations, type checking, strong typing, scoping rules, and referencing environment.

2.2.1 NAMES: Variables names are the most common names in programs.

- Most variables have names.

2.2.2 ADDRESS: The **address** of a variable is the memory address with which it is associated.

- This association is not as simply as it may at first.
- Many languages it is possible for the same name to be associated with different addresses at different places and at different times in the program.
- For example

  A program can have two subprograms, sub1, sub2, each of which defines a local variable that uses the same name, say sum. Because these two variables are independent of each other, a reference to sum in sub1 is unrelated to a reference to sum in sub2. Similarly, most languages allows the same variable to be associated with different addresses at a different times during program execution.

2.2.2.1 ALIASES: It is possible to have multiple variables that have the same address. When more than one variable can be used to access the same memory location, the names are called **aliases.**

- A hindrance to readability because it allows a variable to have its value changed by an assignment to a different variable.
- For example

  The variables total and sum are aliases, any change to total also changes sum and vice versa. A reader of the program must always remember that total and sum are different names for the same memory cell. There can be any number of aliases in a program; this is very difficult in practice.
- Aliasing also makes program verification more difficult.
- Aliases can be created in programs in several different ways.

- Aliases can be created using the data structures in some programming languages, for example, the unions types of C & C++.
- The variable's names are aliases. This and other characteristics of pointers and references.
- Aliasing can be created in many languages through programming parameters.
- The time when a variable becomes associated with an address is very important to an understanding of programming languages.

2.2.3 TYPE: The **type** of a variable determines the range of values the variable can have and the set of operations that are defined for values of the type. For example the **int** type in java specifies a value range of range -2147483648 to 2147483647, and arithmetic operations for additions for addition, subtraction, multiplication, division, and modulus.

2.2.4 VALUE: The value of a variable is the contents of the memory cell or cells associated with the variable. It is convenient to think of computer memory in terms of abstract cells, rather than physical cells. The physical cells, or individually addressable units, of most contemporary computer memories are byte-sized, with a byte usually being 8 bits of length. The size is too small for most program variables. We define an abstract memory cell to have the size required by the variable with which it is associated. For example, although floating-point values may occupy four physical bytes in a particular implementation of a particular language, we think of a floating point value as occupying a single abstract memory cell.

2.3 CONCEPT OF BINDING:

In a general sense, a **binding** is an association, such as between an attribute and an entity or between an operation and a symbol. The time at which a binding takes place is called **binding time**. Binding times are prominent concepts in the semantics of programming languages. Bindings can take place at language design time, language implementation time, compile time, load time, link time, or run time.

- For example,

  The asterisk symbol (*) is usually bound to the multiplication operation at language design time.

- A data type, such as int in C, is bounded to a range of possible values at language implementation time.
- At compile time, a variable in a java program is bound to a particular data type.

- A variable may be bounded to a storage cell when the program is loaded into memory.
- Consider the following C assignment statement, whose variable *count* has been defined as shown:

  int *count*;

  ……….

  count = count + 5;

  Some of the bindings and their binding times for the parts of this assignment statement are as follows:

  - The type of count is bound at compile time.
  - The set of possible values of count is bound at compile design time.
  - The meaning of the operator symbol + is bound at acompile time, when the types of its operands have been determined.
  - The internal representation of the literal 5 is bound at compiler design time.
  - The value of count is bound at execution time with this statement.

  A complete understanding of the binding times for the attributes of program entities is a prerequisite for understanding the semantics of a programming language.

2.3.1 Binding of Attributes to Variables

- A binding is ***static*** if it first occurs before run time and remains unchanged throughout program execution.
- The binding first occurs during run time or can change in the course of program execution, it is called ***dynamic.***
- The physical binding of a variable to a storage cell in a virtual memory environment is complex, because the page or segment of the address space in which the cell resides may be moved in and out of memory many times during the program execution.

2.3.2 Type Bindings

Before a variable can be referenced in a program, it must be bound to a data type. The two important aspects of this binding are how the type is specified and when the binding takes place. Types can be specified some form of explicit or implicit declaration.

2.3.2.1 Variable Declarations

An ***explicit declaration*** is a statement in a program that lists variable names and specifies that they are a particular type. An ***implicit declaration*** is a means of associating variables with types through default conversions instead of declaration statements. Both explicit and implicit declarations create static bindings to types.

- Most of the programming languages designed since the mid-1960s require explicit declarations of all variables (Perl. JavaScript, and ML are three exceptions).
    - Several widely used languages whose initial designs were done before the late 1960s. notably Fortran. PL I. and basic declarations.
    - The implicit declarations can be detrimental to reliability because they prevent the compilation process from detecting some typographical and programmer errors.
- In C and C++, one must sometimes distinguish between declarations and definitions. Declarations specify types and their attributes but do not cause allocations of storage.
- This idea carries over to the functions in C and C++. where prototypes declare names and interfaces. but not the code of functions. Functions definitions. on the other hand. are complete.

2.3.2.2 Dynamic Type Binding

Dynamic type binding, the type is not specified by a declaration statement, nor can it be determined by the spelling of its name.

- ❖ The variable is bound to a type when it is assigned a value in an assignment statement.
- ❖ The assignment statement is executed; the variable being assigned is bound to the type of the value of the expression on the right side of the assignment.

- ❖ Languages in which types are dynamically bound are dramatically different from those in which types are statically bound.
- ❖ The dynamic binding of variables to types is that it provides a great deal of programming flexibility.
- ❖ For example. A program to process a list of numeric data in a language that uses the dynamic type binding can be written as a generic program, meaning that it is capable of dealing with data of any numeric type.
- ❖ Whatever type data is input will be acceptable. because the variables in which the data is to be stored can be bound to the correct type when the data is assigned to the variables after input.
- ❖ In JavaScript and PHP, the binding of a variable to a type is dynamic. For example, a JavaScript may contain the following statement

```
list = [10.2, 3.51]
```

- ❖ There are two disadvantages of dynamic type binding. First, the error detection capability of the compiler is diminished relative to a compiler for a language with static type bindings, because any variable can be assigned a value of any type.

2.3.2.3 Type Inference

- ➢ Meta language is a language that supports both functional and imperative programming. ML employs an interesting type inference mechanism, in which the types of most expressions can be determined without requiring the programmer to specify the types of the variables.
- ➢ **fun** circumf(r) = 3.14159 * r * r;

specifies a function that takes a floating-point argument and produces a floating-point results. The types are inferred from the type of the constant in the expression.

- ➢ **fun** times10(x) = 10*x;

the argument and functional value are inferred to be of type **int.**

- ➢ **fun** square(X) = X * X;

ML determines the type of both the parameter and the return value from the * operator in the function definition. Because this is an arithmetic operator, the type of the parameter and the function are assumed to be numeric. In ML, the default numeric type to be **int.** So, it is inferred that type of the parameter and return value of the square is **int**.

- Type inference is also used in the purely functional language.

2.3.3 Storage Bindings and Lifetime

The fundamental character of an imperative programming language is in large part determined by the design of the storage bindings for its variables.

- The memory cell to which a variable is bound somehow must be taken from a pool of variable memory. This process is called **allocation.**
- **Deallocation** is the process of placing a memory cell that has been unbound from a variable back into the pool of available memory.
- The **lifetime** of a variable is the time during which the variable is bound to a specific memory location.
- The lifetime of available begins when it is bound to a specific cell and ends when it is unbound from that cell.
- We call these categories static, stack-dynamic, explicit heap-dynamic, and implicit heap-dynamic.

2.3.3.1 Static Variables

Static variables are those that are bound to memory cells before program execution begins and remain bound to those same memory cells until program execution terminates.

- Globally accessible variables are often used throughout the execution of a program, thus making it necessary to have them bound to the same storage during that execution
- Advantage of static variables is efficiency. All addressing of static variables can be direct; other kinds of variables often require indirect addressing, which is slower.
- One disadvantage of static binding to storage is reduced flexibility; in particular, in a language that has only variables that are statically bound to storage, recursive subprograms cannot be supported.

- C and C++ allow programmers to include the static specifier on a variable definition in a function, making the variables it defines static.
- Note that when the static modifier appears in the declaration of a variable in a class definition in C++, Java, and C#.

2.3.3.2 Stack-dynamic Variables

Stack-dynamic Variables are those whose storage bindings are created when their declaration statements are elaborated, but whose types are statically bound.

- ❖ **Elaboration** of such a declaration refers to the storage allocation and binding process indicated by the declaration, which takes place when execution reaches the code to which the declaration is attached.
- ❖ For example, the variable declarations that appear at the beginning of a variables defined by those declarations are deallocated when the method completes its execution.
- ❖ Their name indicates, stack-dynamic are allocated from the run-time stack.

Scope :-

. One of the most important factors in gaining an understanding of variables is scope. The scope of a variable is the range of statements in which the variable is visible. A variable is visible in a statement if it can be referenced in that statement.

* The scope rules of a language determine how a particular occurrenc of a name is associated with a variable.

* Scope rules determine how references to variables declared outside the currently executing subprogram or block are associated with their declarations and thus their attributes.

Static Scope :-

ALGOL 60 introduced the method of binding names to nonlocal variables, called static scoping, which has been copied by many subsequ -ent imperative languages, and many nonimperative languages as well,

* In which subprograms cannot be nested, which creates nested static scopes, and those in which subprograms cannot be nested.

* Ada, JavaScript, and PHP allow nested subprograms, but the C-based languages do not.

* The static scoping in this chapter focuses on those languages that allow nested subprograms.

* A program written in a static-scoped language finds a reference to a variable, the attributes of the variable can be determined by finding the statement in which it is declared.

* Static-scoped languages with nested subprograms, this process can be thought of in the following way.

* Suppose a reference is made to a variable 'x' in subprogram Sub1.

* Correct declaration is found by first searching the declarations of subprogram Sub1.

* The search continues in the declarations of the subprogram that declared subprogram Sub1, which is called its "static parent".:

```
procedure Big is
 x : Integer;
procedure sub1 is
 x : Integer;
begin -- of sub1
---
end; -- of sub1
procedure sub2 is
 begin -- of sub2
 --- x ---
end; -- of sub2
begin -- of Big
---
end; -- of Big
```

Blocks :-

* Many languages allow new static scopes to be defined in the midst of executable code.
* This powerful concept, introduced in ALGOL 60, allows a section of code to have its own local variables whose scope is minim- -ized.
* So, they have their storage allocated when the section is entered and deallocated when the section is exited.
* such a section of code is called a "block".

In Ada, blocks are specified with declare clauses, as in

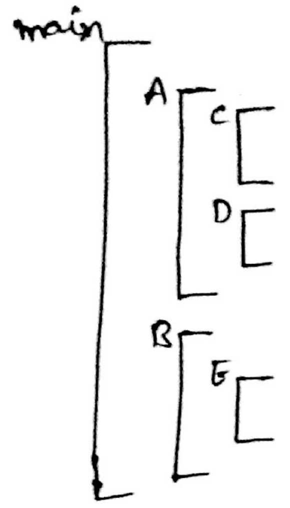
```
 ---
 declare Temp : Integer;
    begin
    Temp := First;
    First := Second;
    Second := Temp;
    end;
 ---
```

Evaluation of Static Scoping :-

* Static scoping provides a method of nonlocal access that works well in many situations. However, it is not without

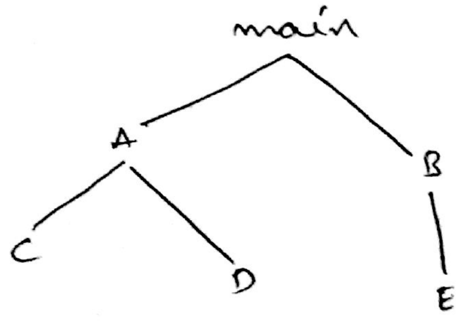
Scanned by CamScanner

Consider the program whose skeletal structure is shown below. For this example, assume that all scopes are created by the definitions of the main program and the procedures.



The structure of a program.

This program contains an overall scope for main, with two procedures that define scopes inside main, A and B. Inside A are scopes for the procedures C and D, Inside B is the scope of produre E. We assume that the necessary data and procedure access determined the structure of the program. The required procedure access is as follows: main can call A and B, A can call C and D, and B can call A and E.



The tree structure of the program.

3 __Dynamic scope__:-

* The scope of variables in APL, SNOBOL4 and the early versions of LISP is dynamic.
* Perl and COMMON LISP also allow variables to the declared to have dynamic scope, although the languages also used Static scoping.
* Dynamic Scoping is based on the calling sequence of subprograms not on their spatial relationship on each other.
* Thus the scope can be determined only at run time.

```
procedure Big is
    x : Integer;
procedure sub1 is
    x : Integer;
    begin   --of sub1
    ----
    end ;   --of sub1
procedure sub2 is
    begin   --of sub2
    ---x---
    end;    --of sub2
begin   --of Big
  ---
  end;  --of ~~sub~~ Big
```

* Assume that dynamic scoping rules apply to nonlocal references.
* The meaning of the identifier x referenced in sub2 is dynamic, it cannot be ~~dynamic~~ determined at compile time.
* It may reference the variable from either declaration of x, dependin -g on the calling sequence.

<u>Evaluation of Dynamic scoping:-</u>

* The effect of dynamic scoping on programming is profound.
* The correct attributes of nonlocal variables visible to a program statement cannot be determined statically. Furthermore, such variables are not always the same.
* Several kinds of programming problems follow directly from dynamic scoping.
* First, during the time span beginning when a subprogram begins its ~~edu~~ execution and ending when that execution ends.
* The local variables of the subprogram are all visible to any other executing subprogram, regardless of its textual proximity or how execution got to the currently executing subprogram.
* There is no way to protect local variables from this accessibility.
* Subprograms are always executed in the environment of all previously called subprograms that have not yet completed their executions.
* A result, dynamic scoping ~~resul~~ results in less reliable programs than static scoping.

* A second problem with dynamic scoping is the inability to statically type check references to nonlocals.
* Dynamic scoping also makes programs much more difficult to read, because the calling sequence of subprograms must be known to determine the meaning of references to nonlocal variables.
* Finally, accesses to nonlocal variables in dynamic-scoped languages take for longer than accesses to nonlocals when static scoping is used.

3(**Scope and Lifetime**:-

* The lifetime of that variable is the period of time beginning when the method is entered and ending when execution of the method termin-ates.
* Although the scope and lifetime of the variable are clearly not the same because static scope is a testual, or spatial, concept whereas lifetime is a temporal concept, they atleast appear to be related in this case.
* Scope and lifetime are also unrelated when subprogram calls are involved Consider the following C++ functions:

```
void printheader() {
  ---
} /* end of printheader */
void compute() {
  int sum;
  ---
  printheader();
} /* end of compute */.
```

*The Scope of the variable sum is completely contained within the compute function.
* It doesnot extend to the body of the function printheader, although printheader executes in the midst of the execution of compute.
* However, the lifetime of sum extends over the time during which printheader executes.)3

Primitive Data Types:-

* Datatypes that are not defined in termsof other types are called primitive data types.
* Nearly all programming languages provide a set of primitive data types.

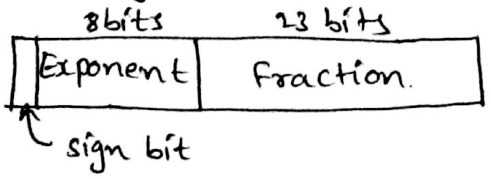
*The primitive types are merely reflections of the hardware. For Example, most integer types.

Numeric types:-

1) Integer:- The most common primitive numeric data types is integer. Many computers now support several sizes of integers. These sizes of integers, and often a few others, are supported by some programming languages. For Example, Java includes four signed integer sizes: byte, short, int and long. Some ~~Java~~ Languages, such as C++ and C#, Include unsigned integer types, which are simply types for integer values without signs. Unsigned types are often used for binary data.

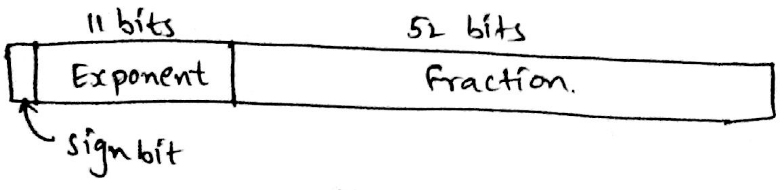
2) Floating-point:- Floating-point data types model real numbers, but the representations are only approximations for most real values. Fundamental numbers π or e (the base for the natural logarithms) can be correctly represented in floating-point notation. Of course, neither of these numbers can be accurately represented in any finite space.

3) Complex:- Some programming languages support a complex data type, for example, Fortran and Python. Complex values are represented as ordered pairs of floating-point values. In Python, the imaginary part of a complex literal is specified by following it with a 'j' or 'J', for example, (7+3j)



(a)

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



(b)

4) Decimal:- Most larger computers that are designed to support business systems applications have hardware support for decimal data types. Decimal data types store a fixed number of decimal digits, with the decimal point at a fixed position in the value. These are the primary data types for business data processing and are therefore essential to COBOL, C# also has a decimal data types.

Decimal types have the advantage of being able to precisely store decimal values, at least those within a restricted range, which cannot be done with floating-point. Decimal types are stored very much like character strings.

using binary codes for the decimal digits. These representations are called binary coded decimal (BCD). In some cases, they are stored on digit per byte, but in others they are packed two digits per byte.

Boolean Types :-

* Boolean types are perhaps the simplest of all types.
* Their range of values has only two elements ; One for true and one for false.
* They were introduced in ALGOL 60 and have been included in most general-purpose languages designed since 1960.
* One popular exception is C89, in which numeric expressions are used as conditionals.
* In such expressions, Boolean types are often used to represent switches or flags in programs.
* Although other types, such as integers, can be used for these purposes, the use of Boolean types is more readable.
* A Boolean value could be represented by a single bit, but because a single bit of memory cannot be accessed afficiently on ~~mod~~ many mach-ines, they are often stored in the smallest efficiently addressable call of memory, typically a byte.

Character Types :-

* Character data are stored in computers as numeric coding.
* The most commonly used coding was the eight-bit code ASCII (Americ-an Standard Code for Information Interchange), which uses the values 0 to 127.
* Because of the globalization of business and the need for computers to communicate with other computers around the world, the ASCII character set is becoming inadequate.
* For example, Unicode includes the Cyrillicalphabet, as used in Serbia, and the Thai digits.

Character String Types :-

A character string types is one in which the values consist of sequences of characters. Characterstring constants are used to label output, and the input and the output of all kinds of data are often done interms of strings.

Design issues:-

The two most important design issues that are specific to character string types are the following:

↳ Should strings be simply a special kind of character array or a primitive type?

↳ Should strings have static or dynamic length?

Strings and Their Operations:-

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. ~~For Example~~

Pattern matching is another fundamental character string operation. In some languages, pattern matching is supported dirrectly ~~colour~~ ~~[illegible]~~ in the language. In others, it is provided by a function or class library.

C and c++ use char arrays to store character strings. These languages provide a collection of string operations through a standard library whose header file is string.h.

The character string literals that are built by the compiler also have the null character. For example, consider the following delarations

```
~~char~~
char str[] = "apples";

strcpy(src, dest);
```

String Length Options:-

There are several design choices regarding the length of string values. First, the length can be static and set when the ~~te~~ string is created. A string is called a static length string. The choice for the strings of Python, the immutable objects of Java's String class, as well as similar classes in the C++ standard class library, Ruby's built-in string class, and the .NET class library available to C#.

The second option is to allow strings to have varying length upto a declared and fixed maximum set by the variable's definition, as exemplified by the strings in C and the C-style strings of C++, these are called limited dynamic length strings.

The third option is to allow strings to have varying length with no maximum, as in JavaScript and Perl. These are called dynamic length strings.

Ada 95 supports all three string length options. Type string from the Standard package provides static length strings.

Array Types:-

An array is a homogeneous aggregate of data elements in which an ~~identifier~~ individual element is identified by its position in the aggregate, relative to the first element.

A reference to an array element in a program often includes one or more nonconstant subscripts.

References require additional run-time calculation to determine the memory location being referenced.

The individual data elements of an array are of some previously defined type, either primitive or otherwise.

Design issues:-

The primary design issues specific to arrays are the followings:

↳ What types are legal for subscripts?.

↳ Are subscripting expressions in element references range-checked?.

↳ When are subscript ranges bound?.

↳ When does array allocation take place?.

↳ Are ragged or rectangular multidimensioned arrays allowed, or both?.

↳ Can arays be initialized when they have their storage allocated?.

↳ What kinds of slices are allowed, if any?.

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.

Symbolically, this mapping can be shown as

array_name(subscript_value_list) → element

The array name is followed by the list of subscripts, which is surrounded by either parentheses or brackets. In most languages that provide multidimensioned arrays as arrays of arrays, each subscript appears in its own brackets.

```
    Sum  := sum + B(I);
```

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:

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.

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. A large array in one subprogram can use the same space as a large array in a different subprogram, as long as both subprograms are not active at the same time.

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

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

A heap-dynamic array is one in which the binding of subscript ranges and storage allocation is dynamic and can change any number of times during the array's lifetime. The advantage of heap-dynamic arrays over the others is flexibility: Arrays can grow and shrink during program execution as the need for space changes. Examples of the five categories are given in the following: ~~paragraphs~~.

Arrays that are declared in C and C++ functions (without the static specifier) are examples of fixed stack-dynamic arrays.

Ada arrays can be stack-dynamic, as in the followings,

```
Get(List_Len);
declare
   List : array (1..List_Len) of Integer;
   begin
   ...
   end;
```

<u>Heterogeneous Arrays</u>:-

A heterogeneous array is one in which the elements need not be of the same type. Such arrays are supported by perl, Python, JavaScript, and Ruby. In all of these languages, arrays are heap dynamic.

In perl, the elements of an arrays can be any mixture of the scalar types, which includes numbers, strings, and references. JavaScript is a dynamically typed language.

<u>Array Initialization</u>:-

Some languages provide the means to initialize arrays at the time their storage is allocated. In Fortran 95, An array can be initialized by assigning it an array aggregate in its declaration. A fortran 95 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/)
```

C, C++, Java, and C# also allow initialization of their arrays, but with one new twist; In the C declaration

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

C and C++ are implemented as arrays of char. These arrays can be initialized to string constants, as in

```
Char name [] = "freddie";
```

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"};
```

5) <u>Associative Arrays</u>:-

An associative array is an unordered collection of data elements that are indexed by an equal number of values called keys. 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.

Each element of an associative array is in fact a pair of entities, a key and a value. Use Perl's design of associative arrays to illustrate this data structure. Associative arrays are also supported directly by Python and Ruby, and by the standard class libraries of Java, C++, and C#.

Structure and Operations:-

In Perl, associative arrays are often 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 must begin with a percent sign (%). Hashes can be set to literal values with the assignment statement, as in

```
%salaries = ("Gary" => 75000, "Perry" => 57000,
             "Mary" => 55750, "Cedric" => 47850);
```

The value parts of hash elements are scalars, so references to hash element values use scalar names. Recall that scalar variable names begin with dollar signs ($). For example,

```
$salaries{"Perry"} = 58850;
```

The delete operator, as in

```
delete $salaries{"Gary"};
```

The empty literal to it, as in

```
@salaries = ();
```

Implementing Associative Arrays:-

The implementation of Perl's associative arrays is optimized for fast lookups, but it also provides relatively fast reorganization when array growth requires it. Each entry and is stored with the entry, although an associative array initially uses only a small part of the hash value. When an associative array must be expanded beyond its initial size, the hash function need not be changed; rather, more bits of the hash value are used. The elements in PHP's arrays are stored in a linked list, in which the nodes appear in the order in which they were created. Links are used to support iterative access to elements through the current and next functions. A hash function is used to provide access to all elements through their keys.

Record Types:

A record is a possibly heterogeneous aggregate of data elements in which the individual elements are identified by names.

There is frequently a need in programs to model collections of data that are not homogeneous. 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 most popular programming languages, except pre-90 versions of Fortran, since the early 1960s, when they were introduced by COBOL.

In C, C++, and C#, records are supported with the struct datatype. In C++, structures are a minor variation on classes. In C#, structures are also related to classes, but are also quite different.

The design issues that are specific to records are:

↳ 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 the homogeneity of elements in arrays versus the possible heterogeneity of elements in records.

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;

```
    01  EMPLOYEE-RECORD.
    02  EMPLOYEE-NAME.
    05  FIRST    PICTURE IS  X(20).
        05 MIDDLE  PICTURE IS  X(10).
        05 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. Any line that is followed by a line with a higher-level number is itself a record. The PICTURE clauses show the formats of the field storage locations, with X(20) specifying 20 alphanumeric characters and 99V99 specifying four decimal digits with the decimal point in the middle.

In Ada, records cannot be anonymous - they must be named types. Consider the following Ada declaration:

```
type Employee_Name_Type is record
    First : String (1..20);
    Middle : string (1..10);
    Last : string (1..20);
end record;
type Employee_Record_Type is record
   Employee_Name : Employee_Name_Type;
   Hourly_Rate : Float;
end record;
Employee_Record : Employee_Record_Type;
```

<u>References to Record fields</u> :-

References to the individual fields of records are syntactically speci-fied by several different methods, two of which name the desired field and its enclosing records. COBOL field references have the form. Where the first record named is the smallest or innermost record that contains the field.

COBOL record example above can be referenced with

MIDDLE OF EMPLOYEE-NAME OF EMPLOYEE-RECORD

Most of the other languages use dot notation for field references, where the components of the reference are connected with periods. For example, the following is a reference to the field Middle in the earlier Ada record example:

Employee_Record.Employee_Name.Middle.

<u>Operations on Records</u> :-

Assignment is a common record operation. In most cases, the types of the two sides must be identical. Ada allows record comparisons for equality and inequality. Also, Ada records can be initialized with aggregate literals. COBOL provides the MOVE CORRESPONDING statement for moving records. This statement copies a field of the specified source record to the destination record only if the destination record has a field with the same name.

This is frequently a useful operation in data-processing applications, where input records are moved to output ~~fiel~~ files after some modification. Because input records often have many fields that have the same names and purposes as fields in output records.

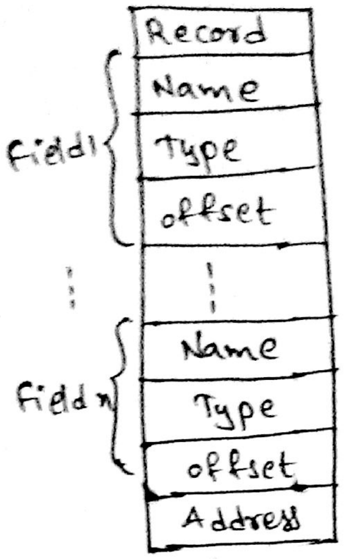
For example, Consider the following COBOL structures:

```
01  INPUT-RECORD.
    02 NAME.
        05 LAST        PICTURE IS X(20).
        05 MIDDLE      PICTURE IS X(15).
        05 FIRST       PICTURE IS X(20).
    02 EMPLOYEE-NUMBER PICTURE IS 9(10).
    02 HOURS-WORKED    PICTURE IS 99.

01  OUTPUT-RECORD.
    02 NAME.
        05 FIRST       PICTURE IS X(20).
        05 MIDDLE      PICTURE IS X(15).
        05 LAST        PICTURE IS X(20).
    02 EMPLOYEE-NUMBER PICTURE IS 9(10).
    02 GROSS-PAY       PICTURE IS 999V99.
    02 NET-PAY         PICTURE IS 999V99.
```

b (**Union Types**:-

A union is a type that may store different type values at different times during program execution. 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:-

↳ Should type checking be required? Note that any such type chec-
-king must be dynamic.

↳ Should unions be embedded in records?

Discriminated Versus Free Unions:-

Fortran, C, and C++ provide union constructs in which there is no language support for type checking. In fortran, the equivalence statement is used to specify unions; in C and C++, it is the union construct.

For Example, Consider the following C union:

```
union flexType {
  int intE1;
  float floatE1;
Union flexType e11;
float x;
...
e11.intE1 = 27;
 x = e11.floatE1;
```

This last assignment is not type checked, because the system cannot determine the current type of the current value of e11, so it assigns the bit string representation of 27 to the float variable x, which of course is nonsense.

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 types 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.

The tag of a constrained variant variable is treated like a named constant. Unconstrained variant records in Ada allow the values of their variants to change types during execution.

The type of the variant can be changed ~~typ~~ only by assigning the entire record, including the discriminant. This disallows inconsistent records because if the newly assigned record is a constant data aggregate, the value of the tag and the type of the variant can be statically checked for consistency.

The following example shows an Ada variant record;

```
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 =>
       Left_Side : Integer;
       Right_Side : Integer;
       Angle : Float;
    when Rectangle =>
      Side_1 : Integer;
      Side_2 : Integer;
  end Case;
end record;
```

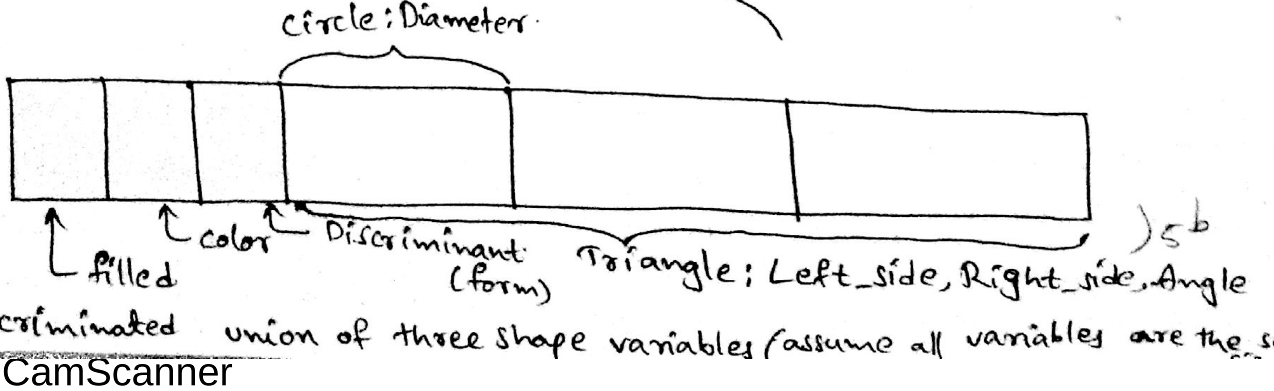

Rectangle : side_1, side_2
Circle : Diameter
filled
color
Discriminant (form)
Triangle : Left_side, Right_side, Angle
5b

fig: A discriminated union of three shape variables (assume all variables are the same

Evaluation:

Unions are potentially unsafe constructs in some languages. They are one of the reasons why Fortran, C, and C++ are not strongly typed; These languages do not allow type checking of references to their unions. On the other hand, unions can be safely used, as in their design in Ada. In most other languages, unions must be used with care.

Implementation of Union Types:

Unions are implemented by simply using the same address for every possible variant. The largest varriant is allocated. In the case of constraine varriants in the Ada language, the exact amount of storage can be used because there is no variation.

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.

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

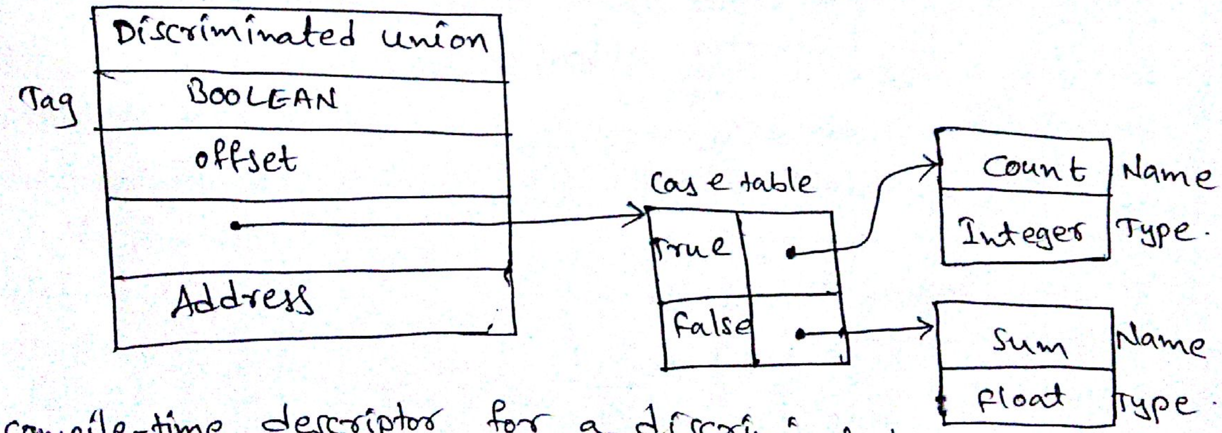


fig: A compile-time descriptor for a discriminated union

Pointer and Reference Types:

A pointer type is one in which the variables have a range of values that consists of memory addresses and a special value, nil. Pointers have been designed for two distinct kinds of uses: first, pointers provide some of the power of indirect addressing, which is heavily used in assem--bly language programming. Second, pointers provide a way to manage dynamic storage. A pointer can be used to access a location in the area

where Storage is dynamically allocated, which is usually called a heap.
Variables that are dynamically allocated from the heap are called heap-dynamic variables. Often do not have identifiers associated with them and thus can be referenced only by pointer or reference type variables. Variables without names are called anonymous variable. Pointers, unlike arrays and records, are not structured types, although they are defined using a type operator (* in C and C++, and access in Ada).

Design issues:-

↳ What are the scope 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 addressi-ng, or both?

↳ should the language support pointer types, reference types, or both?

Pointer Operations:-

Languages that provide a pointer type usually include two fundamen-tal pointer operations: assignment and dereferencing. The first operation sets a pointer variable's value to some useful address. If pointer variables are used only to manage dynamic storage, the allocation mechanism, whether by operator or built-in subprograms, serves to initialize the pointer variable.

A pointer variable in an expression can be interpreted in two distinct ways. First, it could be interpreted as a reference to the contents of the memory cell to which it is bound, which in the case of a pointer is an address.

The pointer could also be interpreted as a reference to the value in the memory cell pointed to by the memory cell to which the pointer variable is bound. The pointer is interpreted as an indirect reference. The former case is a normal pointer reference. The latter is the result of dereferencing the pointer. In C++, it is explicitly specified with the asterisk (*) as a prefix unary operator. If ptr is a pointer variable with the value 7080 and the cell whose address is 7080 has the value 206, then the assignment'

j = *ptr.

sets 'j' to 206. This process is shown below:

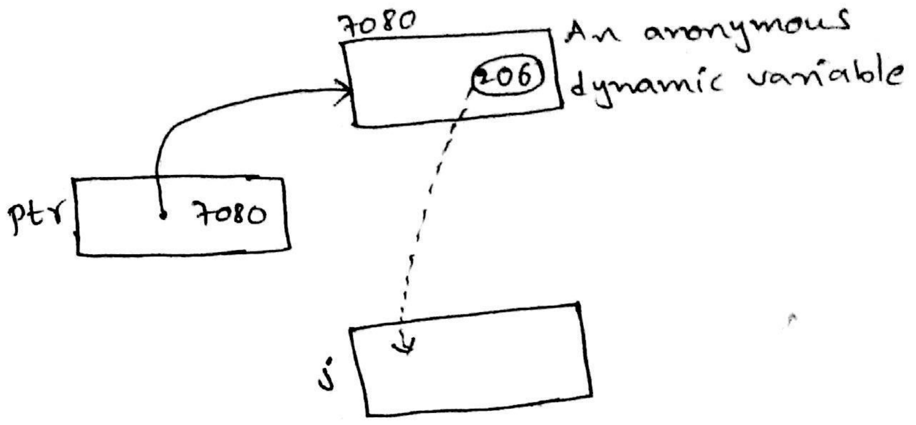


fig:- The assignment operation j = *ptr

Pointer Problems:-

The first high-level programming language to include pointer variable was PL/I, in which pointers could be used to refer to both heap-dyna-mic variables and other program varriables. The pointers of PL/I were highly flexible, but their use could lead to several kinds of programming errors. Some of the problems of PL/I pointers are also present in the pointers of subsequent languages.

Dangling Pointers:-

A dangling pointer, or dangling reference, is a pointer that contains the address of a heap-dynamic variable that has been deallocated. Dangling pointers are dangerous for several reasons. first, the location being pointed to may have been reallocated to some new heap-dynamic variable.

The new variable is not the same type as the old one, type checks of uses of the dangling pointer are invalid. If the new dynamic variable is the same type, its new value will have no relationship to the old pointer's dereferenced value.

Sequence of operations creates a dangling pointer in many langu-ages.

1) Pointer P1 is set to point at a new heap-dynamic variable.
2) Pointer P2 is assigned P1's value.
3) The heap-dynamic variable pointed toby P1 is explicitly deallocated (possibly setting P1 to nil), but P2 is not changed by the operation.

P2 is now a dangling pointer. If the deallocation operation did not change P1, both P1 and P2 would be dangling.

For example, in C++ we could have

```
int *arrayPtr1;
int *arrayPtr2=new int[100];
arrayPtr1 =arrayPtr2;
delete [] arrayPtr2;
//Now, arrayPtr1 is dangling, because the heap storage to
// which it was pointing has been deallocated.
```

In C++, both arrayPtr1 and arrayPtr2 are now dangling pointers, because the C++ delete operator has no effect on the value of its operand pointer. In C++, it is common (and safe) to follow a delete operator with an assignment of zero, which represents null, to the pointer whose pointed-to value has been deallocated.

Arithmetic Expressions:-

Automatic evaluation of arithmetic expressions similar to those found in mathematics, Science, and engineering was one of the primary goals of the first high-level programming languages.

Design issues:-

- ↳ what are the operator precedence rules?.
- ↳ what are the operator associativity rules?.
- ↳ what ~~are~~ is the order of operand evaluation?.
- ↳ Are there restrictions on operand evaluation side effects?.
- ↳ Does the language allow user-defined operator overloading?.
- ↳ What type mixing is allowed in expressions?.

Operator Evaluation Order:-

The language rules that specify the order of evaluation of operators:

1) precedence:-

The value of an expression depends at least in part on the order of evaluation of the operators in the expression. Consider the following expression: a+b*c

Suppose the variables a, b and c have the values 3, 4 and 5 respectively. If evaluated left to right (The addition first and then on multiplication), the result is 35. If evaluated right to left, the result is 23.

The operator precedence rules for expression evaluation define the order in which the operators of different precedence levels are evalu-ated.

Many languages also include unary versions of addition and subtraction. Unary ~~op~~ addition is called the identity operator because it usually has no associated operation and thus has no effect on its operand.

For example,

A + (-B) * C

is legal, but

A + -B * C

usually is not.

Next, consider the following expressions:

-A / B

-A * B

-A ** B

The precedences of the arithmetic operators of a few common program-ming languages are as follows:

	Ruby	C-based Languages	Ada
Highest	**	postfix ++, --	**, abs
	unary +, -	prefix ++, --, unary +, -	*, /, mod, rem
	*, /, %	*, /, %	unary +, -
Lowest	binary +, -		binary +, -

The ** operator is exponentiation. The % operator of the C-based lan-guages and Ruby is exactly like the rem operator of Ada.

2) Associativity :-

Consider the following expression:

a - b + c - d

If the addition and subtraction operators have the same level of precedence, as they do in programming languages, the precedence rules say nothing about the order of evaluation of the operators in this expr--ession.

When an expression contains two adjacent occurrences of operators with the same level of precedence. The question of which operator is evaluated first is answered by the associativity rules of the language.

Associativity in common imperative languages is left to right, except that the exponentiation operator (when provided) associates right to left. In the Java expression

a-b+C

But exponentiation in fortran and Ruby is right associative, so in the expression

A** B **C

In Ada, exponentiation is nonassociative, which means that the expression A**B ** C

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

(A** B) ** c

or

A ** (B ** c)

The associativity rules for a few common imperative langua--ges are given here;

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

3) Parentheses ;-

Programmers can alter the precedence and associativity rules by placing parentheses in expressions. A parenthesized part of an expression

[illegible]

In the expression, (A+B)*C , the addition will be evaluated first. Mathematically, this is ~~[illegible]~~ perfectly natural. In this expression, the first operand of the multiplication operator is not available until the addition in the parenthesited subexpression is evaluated.

4) <u>Conditional Expressions</u> :-

The ternary operator, ?:, which is included in the C-based languages. This operator is used to form conditional expressions.

Sometimes if-then-else statements are used to perform a conditional expression assignment. For example, consider

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

In the C-based languages, this code can be specified more conveniently in an assignment statement using a conditional expression, which has the form

expression_1 ? expression_2 : expression_3 .

<u>Overloaded Operators</u> :-

Arithmetic operators are often used for more than one purpose. For example, in the imperative programming languages + is used to specify integer addition and floating-point addition. Some languages, Java, for example, also use it for string catenation. This multiple use of an operator is called operator overloading and is generally thought to be acceptable, as long as readability and/or reliability do not suffer.

As an example of the possible dangers of overloading, consider the use of the ampersand (&) in C. As a binary operator, it specifies a bitwise logical AND operation. A unary operator, however, its meaning is totally different. As a unary operator with a variable as its operand, the expression value is the address of that variable. The address of operator is the ampersand in this case. For example, the execution of x = &y; causes the address of y to be placed in x.

There are two problems with this multiple use of the ampersand. First, using the same symbol for two completely unrelated operations is detrimental to readability. Second, the simple keying error of leaving out the first operand for a bitwise AND operation can go undetected by the compiler, because it is interpreted as an address-of operator. Such an error may be difficult to diagnose.

Virtually all programming languages have a less serious but similar problem, which is often due to the overloading of the minus operator. The problem is only that the compiler cannot tell if the operator is meant to be binary or unary. So once again, failure to include the first operand when the operator is meant to be binary cannot be detected as an error by the compiler.

Distinct operator symbols not only increase readability, but they are sometimes convenient to use for common operations as well. The floating-point average is to be computed and placed in the floating-point variable avg, this computation could be specified in C++ as

avg = sum / count;

Type Conversions :-

Type conversions are either narrowing or widening. A narrowing conversion converts a value to a type that cannot store even approxi-mations of all of the values of the original type, for example, converting a double to a float in Java (the range of double is much larger than that of float). Example, converting an int to a float in Java. Widening conversions are nearly always safe, meaning that the magnitude of the converted value is maintained.

For example, if the floating-point value 1.3E25 is converted to an integer in a Java program, the result will be only distantly related to the original value.

(The narrowing conversions are:

short to byte or char

char to byte or short

int to byte, short, or char

long to byte, short, char, or int

float to byte, short, char, int, or long

double to byte, short, char, int, long, or float) 6;

Coercion in Expressions:-

One of the design decisions concerning arithmetic expressions is whether an operator can have operands of different types. The compiler chooses the correct type of operation on the basis of the types of the operands. The two operands of an operator are not of the same type and that is legal in the languages, the compiler must choose one of them to be coerced and supply the code for that coercion.

As a simple illustration of the problem, consider the following Java code:

```
int a;
float b,c,d;
...
d=b*a;
```

6) ## Explicit Type Conversion:-

Most languages provide some capability for doing explicit conversions, both widening and narrowing. In some cases, warning messages are produced when an explicit narrowing conversions results in a significant change to the value of the object being converted. In the C-based languages, explicit type conversions are called casts.

(int) angle.

In Ada, the casts have the syntax of function calls. for example

Float(sum) 6a

Errors in Expressions:-

A number of errors can occur in expression evaluation. If the language requires type checking, either static or dynamic, then operand type errors cannot occur. we already discussed the errors that can occur because of coercions of operands in expressions. The other kinds of errors are due to the limitations of computer arithmetic and the inherent limitations of arithmetic. The most common error occurs when the result of an operation cannot be represented in the memory cell where it must be stored. This is called overflow or underflow, depending on whether the result was too large or too small.

Relational and Boolean Expressions:-

Relational Expressions:-

A relational ~~Expressi~~ operator is an operator that compares the values of its two operands. A relational expression has two operands and one relational operator. The value of a relational expression is Boolean, except when Boolean is not a type included in the language.

· The operation that determines the truth or falsehood of a relational expression depends on the operand types.

operation	Ada	c-base languages	fortran 95
Equal	=	==	.EQ. or ==
Not equal	/=	!=	.NE. or <>
Greater than	>	>	.GT. or >
Less than	<	<	.LT. or <
Greater than or equal	>=	>=	.GE. or >=
Less than or equal	<=	<=	.LE. or >=

Boolean Expressions:-

Boolean expressions consist of Boolean variables, Boolean constants, relational expressions, and Boolean operators. Operators usually include those for the AND, OR, and NOT operations, and some times for exclusive OR and equivalence.

Assignment Statements:-

As we have previously stated, the assignment statement is one of the central constructs in imperative languages. It provides the mechanism by which the user can dynamically change the bindings of values to variables.

Simple assignments:-

The general syntax of the simple assignment statement is

<target_variable><assignment_operator><expression>

Nearly all programming languages currently being used the equal sign for the assignment operator. All of these must use something different from an equal sign for the equality relational operator to avoid confus-ion with their assignment operator.

ALGOL 60 pioneered the use of := as the assignment operator, which avoids the confusion of assignment with equality. Ada also uses this assignment operator.

The assignment operator in the C-based languages is treated much like a binary operator.

Conditional Targets:-

C++ allows conditional targets on assignment statements.

For example, consider,

```
flag ? count1 : count2 = 0;
```

which is equivalent to

```
if(flag)
  count 1=0;
else
  count 2=0;
```

Compound assignment operators:-

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

```
a=a+b.
```

Compound assignment operators were introduced by ALGOL 68, were later adopted in a slightly different form by C, the syntax of these assignment operators is the catenation of the desired binary operator to the = operator. For Example,

```
Sum += value;
```

is equivalent to

```
Sum = Sum + value;
```

Unary Assignment Operators:-

The C-based languages, Perl, and Java Script include two special unary arithmetic operators that are actually abbreviated assignments. Combine increment and decrement operations with assignment. Operators ++ for increment, and -- for decrement, used either in expressions or to form stand-alone single-operator assignment statements, can appear either as prefix operators, meaning that they precede the operands, postfix operators, meaning that they follow the operands. In the assignment statement

```
sum = ++ count;
```

The value of count is incremented by 1 and then assigned to sum. This operation could be stated as

```
count = count + 1;
sum = count;
```

If the same operator is used as a postfix operator, as in

```
Sum = count ++;
```

The assignment of the value of count to sum occurs first; then count is incremented. The effect is the same as that of the two statements:

```
sum = count;
count = count + 1;
```

Assignment as an Expression:-

In the c-base languages, Perl, and JavaScript, the assignment statement produces a result, which is the same as the value assigned to the target. It can therefore be used as an expression and as an operand in other expressions. This design treats the assignment operator much like any other binary operator, except that it has the side effect of changing its left operand. For example, in c, it is common to write statements such as

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

In this statement, the next character from the standard input file, usually the keyboard, is gotten with getchar and assigned to the variable ch. Result, or value assigned, is then compared with the constant EOF.

List Assignments:-

Several recent programming languages, including Perl and Ruby, provide multiple-target, multiple-source assignment statements. for example in Perl one can write.

($first, $second, $third) = (20, 40, 60);

The semantics is that 20 is assigned to $first, 40 is assigned to $second, and 60 is assigned to $third. If the values of two variables must be interchanged, this can be done with a single assign-ment, as with

($first, $second) = ($second, $first);

This correctly interchanges the values of $first and $second, without the use of a temporary variable (at least one created and managed by the programmer).

Mixed-mode Assignment:-

We discussed mixed-mode expressions in previous sections. Freq-uently, assignment statements also are mixed-mode. ~~The design~~

Fortran, C, C++, and Perl use coercion rules for mixed-mode assig-nment that are similar to those they use for mixed-mode expression that is, many of the possible type mixes are legal, with coercion freely applied. Ada does not allow mixed-mode assignment.

In a clear departure from C++, Java and C# allow mixed-mode assignment only if the required coercion is widening. An int value can be assigned to a float variable, but not vice versa. The possible mixed-mode assignments is a simple but effective way to increase the reliability of Java and C#, relative to C and C++. That allow mixed-mode assignment, the coercion takes place only after the right-side expression has been evaluated. Alternative woulld be to coerce all operands in the right side to the type of the target before evaluation.

```
Example code :  int a,b;
                float c;
                ...
                c=a/b;
```

Selection Statements:-

A selection statements provide the means of choosing between two or more execution paths in a program. Such statements are fundament-al and essential parts of all programming languages, as was proven by Bohm and Jacopini.

Two-way selection statements:-

Although the two-way selection statements of contemporary imperative languages are quite similar, there are some variations in their designs. The general form of a two-way selector is as follows

```
if control_expression
    then clause
    else clause
```

Design Issues:-

The design issues for two-way selectors can be summerized as follows:
↳ what is the form and type of the expression that controls the selection?.
↳ How are the then and else clauses specified?.
↳ How should the meaning of nested selectors be specified?.

The Control Expression:-

Control expressions are specified in parentheses if the then reserved word (or some other synatactic marker) is not used to introduce the then clause.

In C89, which did not have a Boolean datatypes, arithmeti expressions were used as control expressions. This can also be done in Python, C99, and C++. However, in those languages either arith-etic or Boolean expressions can be used. In other contempora languages, such as Ada, Java, Ruby, and C#, only Boolean expressio can be used for control expressions.

Clause form:-

In many contemporary languages, the then and else clauses appear as either single statements or compound statements. One variation of this is Perl, in which all then and else clauses must be compound statements, even if they contain single statements.

for example,
```
if x>y :
   x=y
   print "case 1"
```

<u>Nesting selectors</u>:

~~Recall~~ the problem of syntactic ambiguity of a straight forward grammar for a two-way selector construct was dissussed as follows:

```
<if_stmt> → if <logic_expr> then <stmt>
          | if <logic_expr> then <stmt> else <stmt>
```

The issue was that when a selection construct is nested in the then clause of a selection construct, it is not clear to which if an else clause should be associated.

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

This construct can be interpreted in two different ways, depending on whether the else clause is matched with the first then clause or the second. Notice that the indentation seems to indicate that the else clause belongs with the first then clause.

<u>Multiple-selection Constructs</u>:-

The <u>multiple-selection</u> construct allows the selection of one of any number of statements or statement groups. A generalization of a selector. In fact, two-way selectors can be built with a multiple selector.

<u>Design issues</u>:-

Some of the design issues for multiple selectors are similar to some of those for two-way selectors.

↳ what is the form and type of the expression that controls the selection?.

↳ How are the selectable segments specified?.

a single selectable segment?.

↳ How are the case values specified?.

↳ How should unrepresented selector expression values be handled, if at all?.

<u>Examples of Multiple Selectors:-</u>

The C multiple-selector construct, <u>switch</u>, which is also part of C++, Java, and JavaScript, is a relatively primitive design. Its general form is

```
Switch (expression) {
        case constant_expression_1: statement_1;
        ...
        case constant_expression_n: statement_n;
        [default: statement_n+1]
    }
```

where the control expression and the constant expressions are some integer type. Selectable statements can be statement sequences, compound statements, or blocks. <u>Default</u> segment is for unrepresen-ted values of the control expression. If the value of the control expression is not represented and no default segment is present, the construct does nothing.

The <u>switch</u> construct does not provide implicit branches at the end of its code segments. This allows control to flow through more than one selectable code segment on a single execution. Consider the following example:

```
switch (index) {
    case 1:
    case 3: odd += 1;
            sumodd += index;
    case 2:
    case 4: even += 1;
            sumeven += index;
    default: printf("Error in switch, index = %d\n",
                                        index);
}
```

This code prints the error message on every execution. Likewise, the code for the 2 and 4 constants is exeuted every time the code at the 1 or 3 constants is executed. The break statement, which is actually a restricted goto, is normally used for exiting switch constructs.

```
switch (index) {
  case 1:
  case 3: odd +=1;
          sumodd += index;
          break;
  case 2:
  case 4: even +=1;
          suneven += index;
          break;
  default :printf(" Error in switch, index = %d \n", index);
}
```

Occasionally, it is convenient to allow control to flow from one selectable code segment to another. This is obviously the reason why there are no implicit branches in the switch construct. The reliability problem with this design arises when the mistaken absence of a break statement in a segment allows control to incorrectly flow to the next segment. The designers of C's switch traded a decrease in reliability for an increase in flexibility.

C's switch is modeled on the multiple-section statement in ALGOL 68, which also does not have implicit branches from selectable segments.

Example of goto:

```
switch (value) {
case -1:
   Negatives ++;
   break;
 case 0:
   zeros ++;
   goto case 1;
case 1:
   positives ++;
default: console.writeline (" error in switch\n"); }
```

Multiple Selection Using if :-

In many situations, a switch or case construct (Ruby's case is an exception) is inadequate for multiple selection.

For example, when selections must be made on the basis of a Boolean expression rather than some ordinal type, nested two-way selectors can be used to simulate a multiple selector.

```
if Count <10 :
  bag1 =True
elif  count <100 :
   bag2 =True
elif  count <1000 :
    bag3 =True.
```

which is equivalent to the following;

```
if count <10 :
  bag1 =True
else :
  if count <100 :
     bag2 =True
  else :
     if count <1000 :
       bag3 =True
     else :
       bag4 =True.
```

Iterative statements :-

An iterative statement is one that causes a statement or collection of statements to be executed zero, one, or more times. An iterative construct is often called a loop. Programming langu-ages from Plankalkul on has included some method of repeating the execution of segments of code. Iteration is the very essence of the power of the computer. If iteration were not possible, programmers would be required to state every action in sequence.

The repeated execution of a statement is often accomplished in a functional language by recursion rather than by iterative constructs.

The first iterative constructs in programming languages were directl
related to arrays. This resulted from the fact that in the earliest
years of the computer era, computing was largely numerical in
nature, frequently using loops to process data in arrays.

Design ~~issues~~ Questions:-

↳ How is the iteration controlled?.

↳ where should the control mechanism appear in the loop constru

Design issues:-

↳ what are the type and scope of the loop variable?.

↳ what value does the loop variable have at loop termination?

↳ should it be legal for the loop variable or loop parameters to be changed in the loop, and if so, does the change affect loop control?.

↳ should the loop parameters be evaluated only once, or once for every iteration?

The Do statements of Fortran 95:-

Fortran 95 has two different counting loop statements, both of which use the Do keyword. The general form of one of the Do statements is

Do label variable = initial, terminal [, stepsize]

where the label is that of the last statement in the loop body, and the stepsize, when absent, defaults to 1. The loop variable must be Integer type; the loop parameters are allowed to be expressions and can have positive or negative values. The stepsize, when present, is not allowed to have the value zero. The loop parameters are evaluated at the beginning of the execution of the Do statement, and the values are used to compute an iteration count, which then has the number of times the loop is to be executed.

Do constructs can be entered only through the Do statement, thereby making the statement a single-entry structure. When a Do terminates - regardless of how it terminates - the loop variable has its most recently assigned value, which the loop terminates.

Consider the construct

```
Do 10 Index = 1,10
...
10 continue.
```

The Ada for statement :-

The Ada for statement has the following form:

```
for variable in[reverse] discrete_range loop
  ...
end loop;
```

A discrete range is a subrange of an integer or enumeration type, such as 1..10 or Monday.. Friday. The discrete range are assigned to the loop variable in reverse order. Note that Ada's for is simpler than Fortran's Do, because the stepsize is always one (or the next element of the discrete range).

The most interesting new feature of the Ada for statement is the scope of the loop variable, which is the range of the loop. The variable is implicitly declared at the for statement and implicitly undeclared after loop termination.

Example :-

```
count : float := 1.35;
for count in 1..10 loop
  sum := sum + count;
end loop;
```

The float variable count is unaffected by the for loop. upon loop termination, the variable count is still float type with the value of 1.35. Also, the float-type variable count is hidden from the code in the body of the loop, being masked by the loop counter count, which is implicitly declared to be the type of the discrete range, Integer.

The for statement of the C-Based languages :-

The general form of C's for statement is

```
for (expression_1; expression_2; exp_3)
    loop body
```

The loop body can a single statement, a compound statement, or a null statement. Because statements in C produce results and thus can be considered expressions, the expressions in a for statement are often statements. The first expression is for initialization and is evaluated only once, when the for statement execution begins. The second expression is the loop control and is evaluated before each execution of the loop body. As is usual in C, a zero value means false and all nonzero values mean true. Therefore, if the value of the second expression is zero, the for is terminated; Otherwise, the loop body statements are executed. In C99, the expression could be a Boolean type. A C99 Boolean type stores only the values 0 or 1. It is often used to increment the loop counter. An operational semantics description of the C for statement is shown next

<u>The for statement of Python:-</u>

The general form of Python's <u>for</u> is

```
for loop_variable in object:
    -loop body
else:
    -else clause
```

The loop variable is assigned the value in the object, which is often a range, one for each execution of the loop body. The else clause, when present, is executed if the loop terminates normally.

Consider the following example:

```
for count in [2, 4, 6]:
    print count
```

<u>produces</u>

```
2
4
6
```

<u>Logically controlled loops:</u>

In many cases, collections of statements must be repeatedly executed, but the repetition control is based on a Boolean expression

[illegible] suppose [illegible] the [illegible]

rather than a counter. For these situations, a logically controlled
loop is convenient: Logically controlled loops are more general than Counter-controlled loops.

Design issues:-

↳ should the control be ~~pretest~~ pretest or posttest?.

↳ Should the logically controlled loop be a special form of a counting loop or a separate statement?

Examples:-

The C-based programming languages include both pretest and posttest logically controlled loops that are not special forms of their counter-controlled iterative statements. The pretest and posttest logical loops have the following forms:

```
while (control_expression)
  loop body
and
do
  loop body
while(control_expression)
```

These two statements forms are exemplified by the following C# code segments:

```
sum = 0;
indat = Int 32. Parse(Console.ReadLine());
 while(indat >= 0) {
   sum += indat;
   indat = Int 32. Parse(Console.ReadLine());
 }
value = Int 32 . Parse (Console.ReadLine());
do {
   value /= 10;
     digits++;
   } while (value >0);
```

The Readline method of the console object gets a line of text from the keyboard. Int 32.parse finds the number in its string parameter, converts it to int type, and returns it.

In the pretest version (while), the statement is executed as long as the expression evaluates to true. In the C, C++, and Java posttest statement (do), the loop body is executed until the expression evaluates to false. The only real difference between the do and the while is that the do always causes the loop body to be executed atleast once.

```
while
 loop:
  if control_expression is false goto out
  [loop body]
   goto loop
  out: ...
  do_while
  loop:
   [loop body]
   if control_expression is true goto loop.
```

User-Located loop control mechanisms:-

In some situations, it is convenient for a programmer to choose a location for loop control other than the top or bottom of the loop. As a result, some languages provide this capability. A syntactic mechanism for user-located loop control, or exit, can be relatively simple so its design is not difficult.

↳ Should the conditional mechanism be an integral part of the exit?.

↳ Should only one loop body be exited, or can enclosing loops also be exited?.

C, C++, Python, Ruby, and C# have unconditional unlabeled exits (break). Java and Perl have unconditional labeled exits (break in Java, last in Perl). Example of nested loops in JAVA, in which there is a break out of the outer loop from the nested loop:

```
Outerloop:
  for (row = 0; row < numRows; row ++)
     for (col = 0; col < numCols; col ++)
```

```
{
    sum+ =mat[row][col];
    if(sum>1000.0)
    break outerLoop;
}
```

~~which~~