A Variable is characterized by a collection of properties or attributes. The most important of these attributes is the type, name, memory, Scope and lifetime.

Names

- The fundamental attribute of a variable is its name. The term identifier is often used for names.
- Names are also associated with functions, parameters and other program constructs.

Design issues

- The primary design issues for names are:
  1) Are names case sensitive?
  2) Are the special words of the language reserved words or keywords?

Name forms

- A name is a string of characters used to identify some entity of a program.
- Fortran 95+ allows up to 31 characters in its name. Most of the ~~names~~ modern languages does not have limit in the number of characters in its name.
- Names in most programming languages have the same form: a letter followed by a string consisting of letters, digits and underscore character(_).

- In C based languages, the underscore has been replaced by "Camel Notation".
- In camel notation, all of the words of a multiple word name except the first are capitalized

Ex: myStack.

- Some languages have special characters at the begining of names.

Ex: PHP - dollar ($)

perl - $, @, %

Ruby - @, @@

- In many languages, notably C-based languages, uppercase and lower case letters in names are distinct; that is names in these languages are case sensitive.

Ex:- rose, ROSE, Rose are three different names.

Special words

- Special words are names given to the actions which are performed in a program.
- Special words can be either reserved or keywords.
- Reserved words are those which cannot be redefined by programmers, but keywords can be redefined.

Ex:- Fortran keyword "Integer."

Integer Apple; ⇒ Integer declaration

Integer = 4; ⇒ Integer is a variable.

- A potential problem with reserved words is that if a language has a large number of reserved words, the user may have difficulty in making up names that are not reserved.

Ex:- COBOL has 300 reserved words.

Variables

- A program variable is an abstraction of a computer memory location or a collection of locations.
- A variable has a set of attributes: name, address, value, type, lifetime and scope.

- Name

- Identifier to distinguish one variable from another. Refer previous topic.

- Address

- The address of a variable is the machine memory address to which it is associated.
- The address is sometimes called the variable's l-value, because the address is what is required when the name of a variable appears in the left side of an assignment.
- It is possible to have multiple variables to have same address. When more than one variable name is used to access the same memory location, the variables are called "aliases."

- Type

- The type of a variable determines the range of values that the variable can store and operate.

Value

- The value of a variable is the contents of the memory location or locations associated with the variable.
- A "abstract memory cell" has the size required by the variable with which it is associated. For example a floating-point value may occupy four physical bytes in a particular implementation, but a floating-point value is thought of as occupying a single abstract cell.
- A variable's value is sometimes called its r-value, because it is what is required when the name of the variable appears on the right side of an assignment.

Binding

- A Binding is an association between attribute and a entity, such as between a variable and ity type or value or between an operation and a symbol.
- The time at which a binding takes place is called binding time.
- A binding can take place during

1) language design time → * for multiplication
2) language implementation time → A data type, int to possible range of values.
3) Compile time → A variable is bound to a particular data type.
4) Load time → A variable is bound to memory location.
5) Link time → A call to subprogram/function is bound to subprogram co
6) Run time → A value of variable is bound during execution.

Ex- count = count + 5;

Binding of attributes to Variables

- A binding is static if it occurs before run time begins and remains unchanged throughout program execution
- If the binding first occurs during run time or can change in the course of program execution, it is called dynamic.

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:
    1) How the type is specified
    2) when the binding takes place.
- Types are specified statically through some form of explicit or implicit

Static type Binding

- An explicit declaration is a statement in a program that lists variable names and specifies that they are a particular type. int test;
- An implicit declaration is a means of associating variables with types through default conventions, rather than declaration statements. Ex: In java script list = [10.2,3.5]; list = 10;
- Both explicit and implicit declarations create static bindings to types.

Dynamic type Binding

- The variable is bound to a type when it is assigned a value in an assignment statement.

- A variable's type can change any number of times during program execution.

- It is [redacted] important to realize that the type of a variable whose type is dynamically bound may be temporary.

- Dynamic type binding was introduced in C# 2010. A variable can be declared to use dyamic type binding by including the dynamic reserved word in its declaration.

```
dynamic any;
```

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 determines how references to variables declared outside the currently executing block are associated with their declarations.
- A variable is local in a program unit or block in which it is declared.
- The non-local variable of a program unit or block are those that are visible within the program unit or block, but are not declared there.
  - Global variables are special category of non-local variables.

Static Scope

- Static scope mean that the scope of a variable can be statically determined i.e prior to execution.
- There are two categories of static scope languages : those in which subprograms can be nested, which creates nested scope and those in

which subprograms cannot be nested.

Ex:

```
function big()
{
    function sub1()
    {
        var x = 7;
        sub2();
    }
    function sub2()
    {
        var y = x;
    }
var x = 8;
sub1();
}
```

⇒ under static scoping, the reference to variable x in sub2 is the x declared in the procedure Big.

→ A reference to a variable can be determined by finding the statement in which it is declared. If no declaration is found, the search continues in the outer function (or) static parent.

Blocks

A Block is a section of code, which have its own local variables, whose scope is confined to the block.

Each block defines a new scope.

C-based languages allow any compound statement to have declarations. Such compound statements are called blocks.

References to the variables in a block that are not declared are connected to the declarations of outer blocks.

Ex:

```
void sub()
{
  int count = 5;
  printf(count);
    while( )
    {
      int count = 10;
      printf(count);
    }
}
```

⇒ Count of sub is hidden to count of while.

Declaration order

- In C and in some other languages, all data declarations in a function except those in nested blocks must appear at the beginning of the function.
- However some languages like C++, Java, JavaScript allow variable declara to appear anywhere.
- Declarations may create scope. For example in C99, C++ and Java, the scope of all local variables is from their declarations to the end of the block.

Ex:

```
main()
{
    printf(count);   // error
    int count
}
```

Global Scope

- Languages such as C, C++, PHP, JavaScript, allow a program structure that is a sequence of function definitions in which variable definitions can appear outside the function.
- Definitions outside the functions create global variables, which can potentially visible to all functions. This is referred to as global scope

```
#include<stdio.h>
  int a;        // global variable.
main()
 {
   print(a);
 }
add()
 {  print(a);
 }
```

Dynamic Scoping

- Dynamic scope refers to the scope of variables and subprograms, which can be determined only at the run time.

```
function big()
{
  function sub1()
    {
    var x = 7;
    }
  function sub2()
    {
    var y = x;
    }
var x = 3
}
```

- Assume that dynamic scoping rules apply to non-local references.
- The meaning of x in sub2 is dynamic - it cannot be determined at compile time.
- perl uses dynamic scoping.

Scope and Lifetime

- Sometimes scope and lifetime of a variable appear to be related.

```
Void print()
  {
  ...
  }
  Void compute()
    {
      int sum;
    } print();
```

- The scope of the variable sum is completely inside the compute function. It does not extend to the body of function print, although print executes inside compute function.
- However the lifetime of sum extends over the time during which print function executes.
- The storage location for sum is bound before the call to print function, will continue its binding during and after the execution of print.
- The lifetime of a variable is the time during which the variable stays in memory and is therefore accessible during program execution.
- The variables, that are local to a method are created the moment, the method is activated and are destroyed when the activation of the method terminates.

Garbage collection

- There are several different approaches to garbage collection.
  1) Reference counters (or) Eager approach
  2) Mark-Sweep (or) Lazy approach.

Reference counters / Eager approach

- In this method of storage reclamation, every memory cell has a counter that stores the number of pointers that are currently pointing to the cell.
- If the reference counter reaches zero, it means that no program pointers are pointing at the cell and it has thus become garbage and can be returned to the list of available space.
- There are three problems with reference counters:
  1) If storage cells are relatively small, the space required for counters is significant.
  2) Some execution time is required to maintain the counter values.
  3) Complications arise when a collection of cells are connected circularly.

Mark-Sweep / Lazy approach

- It consists of three phases:
  1) Every cell in dynamic memory (heap) have indicators set to indicate they are garbage.
  2) The run time system allocates storage as requested and the alloted cells are marked as not being garbage. This is marking phase.

3) The third phase is sweep phase in which all the cells, that have not been marked as still being used are returned to the list of available space.

- The original mark-sweep was done too infrequently, so due to this applications had less storage for allocation. To avoid this problem incremental mark-sweep, is used in which garbage collection occurs more frequently, so the storage space is available for application.

Type checking

- It 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 implicity converted to a legal type. This automatic conversion is called coercion.

  Ex:- Addition of an int and float variable in Java, involves coercion of int variable to float-point.

- A type error is the error which would happen, when an operator is applied with operands of inappropriate type.
- Type checking can be static or dynamic based on the time it is performed.
- Static type checking is preffered because, earlier correction is usually less costly.
- JavaScript, PHP supports dynamic type checking.

Datatypes

- A datatype defines a collection of data values and a set of predefined operations on those values.
- Data types available in the language have to match the objects in the real-world of the problem being addressed.

There are number of uses of the type system of a programming language:

1) Error detection - Type checking is directed by the type system of a language.
2) program modularization - user defined and derived data types provides program modularization. Abstract data types enclose the operations and functions.
3) Documentation :- Information about the data provides the clues about the program behaviour.

- The type system of a programming language defines how a type is associated with each expression in the language and includes its rules for type equivalence and type compatibility.

primitive data types

- Data types that are not defined in terms of other types are called primitive data types. The various types are:

1) Numeric types.
2) Boolean types
3) Character types.

Numeric types

- Many early programming languages had only numeric primitive types.

1) Integer

- The most common primitive numeric data types is integer.
- Many computers now support several sizes of integers.

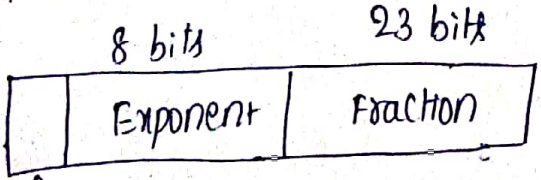
  Ex: Java supports byte, short, int and long.

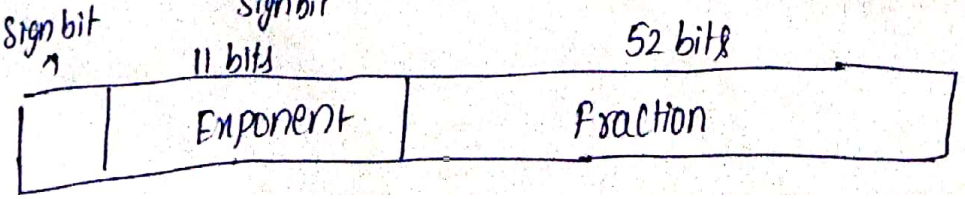
  C++ includes unsigned integer types.

- Most computers use two's complement notation to represent the negative numbers. Some computers use one's complement, but the disadvantage is that in one's complement there are two representations for zero.

Floating-point

- Floating-point data types model real numbers. Floating point numbers are stored in binary.
- Most languages include two floating-point types, float and double. The float type is the standard size, usually stored in four bytes. The double type is provided for situations where larger fractional parts and larger range of exponents is needed.
- The floating point formats are:





complex

- Some programming languages support a complex data type.
- complex values are represented as ordered pairs of floating-point values.

Ex: In python (7+3j)

- Languages that support a complex type includes operation for arithmetic on complex values.

Decimal

- Many larger computers that support business system 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.
- Decimal types are stored very much like character strings, using binary code for decimal digits. These representations are called binary coded decimal (BCD).

Boolean types

- Boolean types are the simplest of all types.
- Their range of values has only two elements : one for true and one for false.
- Boolean types are often used to represent switches or flags in the programs.

Character type

- character data are stored in computers as numeric codings.
- the most commonly used coding was 8-bit code ASCII, which uses the values 0 to 127 to code 128 different characters.

- To increase the number of characters, a new character code called Unicode was introduced, which is a 16-bit character set.
- To provide the means for processing codings of single characters, most programming languages include a primitive type for them.

Character string types

- A character string type is one in which the value consist of sequences of characters.
- The two most important design issues that are specific to character string types are:
    1) Should strings be simply a special kind of character array or a primitive type?
    2) Should strings have static or dynamic length?

Strings and their operations

- The most commong string operations are assignment, catenation, substring reference, comparison and patter match.
- If strings are not defined as a primitive type, string data is usually stored in arrays of single characters and referenced as such in the language.
- C and C++ use char arrays to store character strings. These languages provide a collection of string operations through standard library functions.

- The common string operations supported by C standard string library are.

String copy → Strcpy (dest, scr);

String length → Strlen (str1);

String concatenation → Strcat (str1, str2).

- Perl, JavaScript, Ruby, PHP includes built-in pattern matching operations.

~~String Types~~

- An example of a pattern matching expression is follows:

/[A-Za-z] [A-Z a-z/d]+/

- The above pattern matches the typical name form in a programming language.
- The brackets enclose character classes.
- The first character class specifies all letter, the second specifies all letters and digits. The plus operator following the second character class specifies there must one or more of the category.

String length options

- There are several design choices regarding the length of string values:

1) The length can be static and set when the string is created. Such a string is called static length string.

Ex: python.

2

2) This option allows strings to have varying length upto a declared and fixed maximum set by the variable's definition. These are called limited dynamic length strings.

Ex:- C, C++ strings.

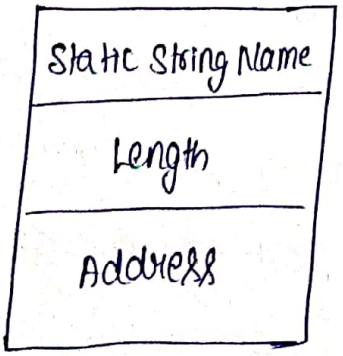
3) The third option is to allow strings to have varying length with no maximum. These are called dynamic length strings.

Ex:- Javascript, perl.

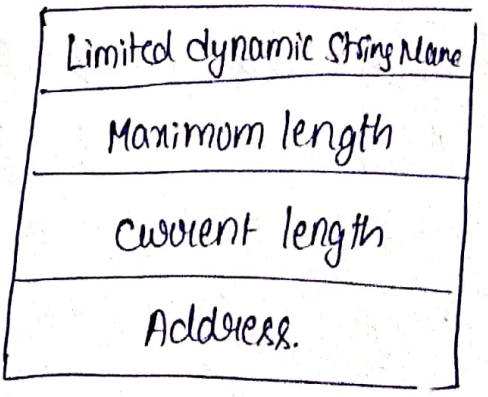
Implementation of character String types

- A descriptor is one which specifies attributes or properties of a variable.
- A descriptor for strings are as follows:

1) For static strings



2) For limited dynamic strings



3) There are three approaches to support dynamic length string allocation and deallocation:

(i) Strings can be stored as linked lists. Memory is assigned from heap.

(ii) Strings are stored as array of pointers to individual characters.

(iii) Store complete strings in adjacent storage cells.

Array types

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

- The individual elements of an array are of the same type.

- References to individual array elements are specified using subscript expressions.

- In many languages, such as C, C++, Java, Ada and C# all of the elements of an array are required to be of the same type. Java 5.0 provides generic arrays.

Design Issues

- The primary design issues specific to arrays are:

1) what types are legal for subscripts.?
2) Are subscripting expressions in element references range checked?
3) when are subscript ranges bound?
4) when does array allocation take place?
5) Are ragged or rectangular multidimensional arrays allowed or both?
6) Can arrays be initialized when they have their storage allocated?
7) what kind of slices are allowed, if any?

1) The type of subscripts is often a subrange of integers. But some languages like Ada allows any type to be used as subscripts such as boolean, character and enumeration.

2) Early programming languages did not specify that subscript ranges must be checked. Range errors in subscripts are common in programs.

- So requiring range checking is an important factor in the reliability of languages.

3) Subscript Binding 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.
- In some languages, the lower bound of the subscript range is implicit. For example, in the C-based languages, the lower bound of all subscript ranges is fixed at 0.

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

1) A static array is one in which the subscript ranges are statically bound and storage allocation is static.

2) A fixed stack-dynamic array is one in which the subscript ranges are statically bound, but the allocation is at declaration elaboration time during execution.

3) A stack-dynamic array is one in which both the subscript ranges and storage allocation are dynamically bound at elaboration time.

- A fixed heap-dynamic array is that 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 life time.

4) Array Initialization

- Some languages provide the means to initialize arrays at the time of their storage is allocated.

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

- In c and c++ arrays can be initialized to string constants.

```
char name[] = "Kishore";
```

- Arrays of strings can also be initialized with string literals

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

- Same initialization in Java

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

5) Array operations

- An array operation is one that operates on an array as a unit.
- The most common array operations are assignment, catenation, comparision for equality and inequality and slices.
- C-based languages does not provide an array operations.
- Python provides array assignment, array catenation (+) and element membership(in).

6) Rectangular and Jagged Arrays

- A rectangular array is a multidimensioned array in which all of the rows have the same number of elements and all of the columns have the same number of elements.
- A jagged array is one in which the lengths of the rows need not be the same.
- C, C++, Java support jagged arrays but not rectangular arrays.
- Fortran, Ada, C# support rectangular arrays.

7) Slices

- A slice of an array is some substructure of that array. Slice is a mechanism for referencing part of an array as a unit.
- Consider the following python declarations:

```
vector = [2, 4, 6, 8, 10, 12, 14, 16]
math = [ [1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

- A python slice is as follows:

vector[3:6] → elements with the subscripts 3, 4 & 5

→ [8, 10, 12].

Associative arrays

- An associative array is unordered collection of data elements that are indexed by an equal number of values called keys.
- In an non-associative arrays, the indices never need to be stored but in an associative array, the user defined keys must be stored in the structure.

- So each element of an associated array is in fact a pair of entities, a key and a value.
- Associated arrays are supported by perl, Python, Ruby and are supported by standard class libraries of Java, C++, C#.

Structure and operations

- In perl associative arrays are called hashes, because in implementation their elements are stored and retrieved with hash functions.
- Every hash variable must begin with a percent sign (%). Each hash element consists of two parts: a key, and a value.

```
%salaries = { "Gary" => 75000, "Perry" => 57000 };
```

- Individual elements are accessed as follows:

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

- An element can be removed from the hash by using delete operator.

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

- The entire hash can be emptied by assigning the empty literal to it

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

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.
- In this the individual elements are not of the same type or size.
- In C, C# and C++ records are supported by struct data type.

- The design issues that are specific to records are:

1) what is the syntactic form of references to fields?

2) Are elliptical references allowed?

```
Struct Book
{
   char * name;
   char * author;
   float price;
};
```

References to fields

- References to the individual fields of records are syntactically specified by several different methods.

1) Most languages use dot notation for field references, where the components of the reference are connected with periods.

```
Ex: Book B1;
    B1.price;
```

2) other languages use a method, in which the name of the name of the desired field and its enclosing records are specified. For example MIDDLE field in the COBOL record is referenced as follows:

```
MIDDLE OF EMPLOYEE-NAME OF EMPLOYEE-RECORD.
```

Elliptical References

- A fully qualified reference to a record field is one in which all

intermediate record names from the largest enclosing record to the specific field are name in reference.

- Some languages allow elliptical references to record fields in which any or all of the enclosing record names can be omitted.

Ex: MIDDLE

MIDDLE OF EMPLOYEE_NAME are elliptical reference.

Union Types

- A union is a type whose variables may store different type values at different times during program execution.
- All the datatypes uses the same memory location during different times during program execution.

Design issues

- Should type checking be required ? Note that any such type checking must be dynamic.
- Should unions be embedded in records?

Free unions Vs Discriminated unions

- C and C++ provide union constructs in which there is no language support for type checking. These unions are called free unions, because programmers are allowed complete freedom type checking in their use.

~~Type checking of~~

Ex:

```
union flexType
{
  int E1;
  float E2;
};
```

```
union flexType *fl;
float x;

fl.E1 = 27;
x = fl.E1;
```

- The last assignment is not type checked. This is allowed in free unions.

- Type checking of unions requires that each union construct include a type indicator. Such an indicator is called a tag or discriminant.

@ A union with a discriminant is called a discriminated union.

- These are supported by Ada,.

```
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;
  end case
end record.
```

Pointer and Reference Type

- 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 are designed for two distinct kinds of uses:
  1) pointer provides a way for indirect addressing.
  2) pointers provide a way to manage dynamic storage, when the memory is dynamically allocated from heap.

- Pointers, unlike arrays and records are not structured types, although they are defined using a type operator.

- They are different from scalar (numeric) variables because they are used to reference some other variable, rather than being used to store data.

- These two categories of variables are called reference types and value types.

Design issues

- The primary design issues related to pointers are:

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

Pointer operations

- Languages that provide a pointer type usually include two fundamental pointer operations: assignment and dereferencing.
- Assignment sets a pointer variable's value to some useful address.
- The second case could also be interpreted as a reference to the value in the memory location points to the value (i.e value at the address of).

- Languages that provide pointers for the management of a heap must include an explicit allocation operation.
- object-oriented languages provide new operators and delete for memory allocation and deallocation.

Reference types

- A reference type variable is similar to a pointer, except that a pointer refers to an address in memory, while a reference refers to an object or value in memory.

Ex:
```
int result = 0;
int &ref_result = result;
ref_result = 100;
```

- In the above code segment, result and ref_result are aliases and points to the same memory location.

Expressions and Assignment Statements

Unit-2

(2)

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

Arithmetic expressions

- Most of the characteristics of arithmetic expressions in programming languages were inherited from mathematics.
- In programming languages, arithmetic expressions consist of operators, operands, parentheses and function calls.
- A operator could be unary, binary or ternary based on the number of operands involved.
- The primary design issues for arithmetic expressions are as follows:
  1) what are the operator precedence rules?
  2) what are the operator associativity rules?
  3) what is the order of operand evaluation?
  4) Are there restrictions on operand evaluation side effects?
  5) Does the language allow user-defined operator overloading?
  6) what type mixing is allowed in expressions?

operator Evaluation order

- The operator precedence and associativity rules of a language dictate the order of evaluation of its operators.

precedence

- precedence is giving priority to some operators than the remaining

ones due to higher level of complexity.

- The operator precedence rules for expression evaluation partially define the order in which the operators of different precedence levels are evaluated. The operator precedence levels is based on the hierarchy of operator priorities, as seen by the language designer.
- The precedence of the arithmetic operators are as follows.

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

- ** operator is for exponentiation.

Associativity

- 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.
- An operator can have either left or right associativity.
- Associativity in most languages is left to right except the exponentiation operator, which sometimes have right to left associativity.

The associativity rules for some common languages are as follows:

Language	Associativity Rule
C-based language	Left to Right: *, /, %, binary +, binary - Right to left: ++, --, unary -, unary +
Ruby	Left to Right: *, /, +, - Right to Left: **

Parentheses

Programmers can alter the precedence and associativity rules by placing parantheses in expressions.

A parenthesized part of an expression has precedence over its adjacent unparanthesized parts.

(A+B) *C

- Addition will be evaluated first.

Operand Evaluation order

- Another design issue to consider is the order of evaluation of operands.

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

If neither of the operands of an operator has side effects, then operand evaluation order is irrelevant.

Sideeffects

A side effect of a function, naturally called a functional side effect, occurs when the function changes either one of its parameters or a global variable.

- consider the following example

```
int a=5;
int fun1()
{
  a=17;
  return 3;
}
Void main()
{
 a= a+ fun1();
}
```

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

- There are two possible solutions to the problem of operand evaluation order and side effects.

1) The language designer could disallow function evaluation from affecting the value of expressions by simply disallowing functional side effects.

2) Second the language definition could state that operands in expressions are to be evaluated in a particular order and demand that the implementors guarantee that order.

Overloaded operators

- Arithmetic operators are often used for more than one purpose. Such use is termed as operator overloading.

  Ex: Ampersand (&) in C++

  - Binary "&" is used as bitwise logical AND operation.
  - Unary "&" is used as address-of operator.

- There are some problems with this use of same operator for multiple purposes:

1) Using the same symbol for two completely unrelated operations has a problem of readability.

2) Errors are hard to be ~~diagnosed~~ diagnosed by the compiler with regard to operator overloading.

- Some languages like C++ support user-defined operator overloading. C++ supports overloading most of the operators except a few like structure member operator (.) and the scope resolution operator (::).

Type Conversions

- Type conversions can be two types:
    1) Narrowing
    2) Widening.
- A Narrowing conversion converts a value to a type that cannot store some parts of the original type.

    Ex: Float to int., double to float
- A widening conversion converts a value to a type that can include at least all parts of the value of the original type.

    Ex: int to float, float to double.

Widening conversions are always safe. Narrowing conversions are not always safe.

Type conversions can be either implicit or explicit

Implicit Type Conversions

One of the design decisions related to arithmetic expressions is whether an operator can have operands of different types.

Languages that allow such expressions are called mixed mode expressions.

Such expressions must define conversions.

"coercion" is an implicit type conversion that is initiated by the compiler.

When two operands of an operator are not of the same type and that is legal in the language, the compiler must choose one of them to be coerced.

Explicit Type conversion

- Most languages provide some capability for doing explicit conversions, both widening and narrowing.
- In C-based languages, explicit type conversions are called casts. To specify a cast, the desired type is placed in parantheses just before the expression to be converted:

```
(int) angle;
```

Errors in Expressions

- A number of errors can occur during expression evaluation.
- 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 underflow or overflow.
- one limitation of arithmetic is that division by zero is disallowed.
- Floating-point overflow, underflow and division by zero are examples of run-time error, which are sometimes called exceptions.

Relational Expressions

- A relational operator is an operator that compares the value 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.

Relational operators in C-based languages are: ==, !=, >, >=, <, <=

Boolean Expressions

Boolean expressions consists of boolean variables, boolean constants, relational expressions and boolean operator.

Boolean operators include AND, OR and NOT operators and sometimes for exclusive OR.

The precedence of the arithmetic, relational and Boolean operators in the C-based language is as follows.

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

Lowest

Assignment Statements

- Assignment statement is one of the main important constructs of a programming language.
- It provides the mechanism by which the user can dynamically change the bindings of values to variables.

i) Simple assignments

- All programming languages use the equal sign for the the assignment operation.

- In some languages, an assignment can appear only as a stand-alone statement and the destination is restricted to a single variable. There are other alternatives.

2) Conditional Targets

- perl allows conditional targets on assignment statements. For example

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

which is equivalent to

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

3) Compound assignment operators

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

a = a+b. is written as a += b;

4) Unary assignment operators

- The C-based languages, perl and javascript include two special unary arithmetic operators. They also combine increment and decrement operations with assignment.

- The operators ++ (increment) and --(decrement) can be used either in expressions or to form stand-alone single-operator assignment statement.

- They can be either prefix operators or post-fix operators.

- In the following assignments

```
Sum = ++count;   ⇒  count = count + 1;  Sum = count;
```

- The value of the count is incremented by 1 and then assigned to Sum.

```
Sum = count++;   ⇒  Sum = count;  count = count + 1.
```

- The value of count is assigned first and the count value is incremented.

- As an unary increment operator, it simply increments the value of the value.

```
count++;
++count;   } → count = count + 1;
```

- When two unary operators apply to the same operand, the association is right to left. For example

  - count++ is evaluated as

  - -(count++).

Assignment as an Expression

- In C-based languages, the assignment statement produces a result that is same as the value assigned to the target. It can be therefore be used as an expression and as an operand in other expressions.

Ex:

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

- Due to this, treatment of assignment operator as any other binary operator, it allows multiple-target assignments

```
sum = count = 0;
```

6) Multiple Assignments

- Several recent programming languages including perl, Ruby provide multiple-target, multiple-source assignment statements.

Ex:-

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

- If two values must be interchanged, this can be done with a single statement

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

Mixed mode Assignment

- Mixed mode expressions are common.
- So, does the type of the expression have to be the same as the type of the variable being assigned or can coercion be used in some cases of mismatch?
- Fortran, C, C++ and perl uses coercion rules for mixed-mode assignment.

Ex:-
```
int x;
int a; float c, d;
x = a + c + d;
```
⇒ a is coerced to float and float value obtained from a+c+d is coerced to int to store in x.

Statement-level control structures

Statements that supports means for selecting among alternative control flow paths and some means of causing the repeated execution of statements or sequence of statements are called control structures.

- A control structure is a control statement and the collection of statements whose execution it controls.

There is only one design issue that is relevant to all of the selection and iteration control statements.

→ Should the control structure have multiple entries?

Selection Statements

A selection statement provides the means of choosing between two or more execution paths in a program.

Selection statements fall into two general categories:

1) two-way selection & n-way selection.
2) multiple selection.

Two-way Selection Statements

- The design issues for two-way selection statements are:
  1) what is the form and type of expression that controls the selection?.
  2) How are the then and else clauses specified?
  3) How should the meaning of nested selections be specified?.

Control expressions

- These are specified in parenthesis of if.

- Arithmetic or Relational or Boolean expressions are used as boolean expressions.

Clause form

- In most of the languages, then then and else clauses appear as either single statements or compound statements.
- Many languages use braces to form compound statements, which serve as the bodies of then and else clauses.

```
if (conditional-expression)
{
  then clause statements;
}
else
{
  else clause statements;
}
```

Nesting

- when a selection statement is nested in the then clause of a selection statement, it is not clear to which if an else clause should be associated.
- Example

```
if (som==0)
  if (coont==0)
    result = 0;
  else
  result = 1;
```

the above statement can be interpreted in two different ways, depending on whether the else clause is matched with the first then clause or the second one.

To avoid this problem, in Java and many other languages, the static semantics of the language specify that the else clause is always paired with the closest unpaired then clause.

To force the alternative, we have to use various methods in various languages.

In Java, we use the braces:

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

- In perl, each clause is compounded.

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

In Ruby, clauses are ended with end statements.

```
if a > b then
  sum = sum + a;
  account = account + 1;
else
  sum = sum - sum16;
  bcount = bcount + 1;
end.
```

⇒

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

Multiple-Selection Statements

- The multiple-selection statement allows the selection of one of any number of statements or statement groups.
- Although multiple selector can be built from the two-way selectors and gotos, the resulting structures are unreliable, difficult to read and write.
- The various design issues are:
  1) what is the form and type of the expression that controls the selection?
  2) How are the selected segments specified
  3) Is execution flow through the structure restricted to include just a single selectable segment?
  4) How are the case values specified.
  5) How should unrepresented selector expression values be handled, if at all?

Multiple Selectors

```
switch (expression)
{
case const_expression1 : statement;
   ⋮
case cont_expression n : statementn
default : statementn+1
}
```

- In this expression and constant expressions can be integer types as well as characters and enumeration types.

- The selectable segments can be statement sequences, compound statements or blocks. ③
- The optional default segment is for unrepresented values of the control expression.
- understand the flow of execution within a switch construct.

```
Ex: Switch ( index )
    {
    case 1:
    case 3 : odd = odd +1;
             Somodd = Somodd + index;
    case 2 :
    case 4 : even = even +1;
             Someven = Someven + index;
    default : printf (" error");
    }
```

- To modify the flow of execution we use break statements.

Multiple selection using If

- The multiple ~~section~~ selection can also implemented using if statements and nesting. The nested selector is called an else-if clause.

```
if (count < 10 )
  bag1 = true;
  else if (count <100 )
      bag2 = true;
    else if ( count < 100 )
        bag3 = true;
      else  bag4 = true;
```

Implementing multiple selection structures

- It can be implemented with multiple conditional branch instructions.

→ switch (expression)
{
case const_exp1 : Statement 1;
break;
.
.
.
case const_exp2 : Statement n;
break;
default : Statement n+1
}

translated into →

→ code to evaluate expression into t
goto branches
label 1 : Statement 1
goto out;
.
.
.
label n : Statement n
goto out;
default : statement n+1 ; goto out;
branches : if t = const_exp1 goto label 1
.
.
.
if t = const_expn goto label n
goto default
out:

- An alternative to these branches is to put the case values and labels in a table and use a linear search with a loop to find the correct label. This would require less space.
- If the number of case labels is greater than 10, the compiler can build a hash table of the labels.

Iterative Statements

- An iterative statement is one that causes a statement or a collection of statements to be executed zero, one or more times.
- An iterative statement is often called a loop.

The basic design issues are:

1) How the iteration controlled?

2) where should the control mechanism appear in the loop statement?

The primary possibilities of iteration control are:

1) Logical

2) ~~counter~~ counting

3) combination of the two.

The main choices for the location of the control mechanism are

1) Top of the loop

2) Bottom of the loop.

The body of an iterative statement is the collection of statements whose execution is controlled by the iteration.

when the test for loop completion occurs before the body of loop is executed, then it is called pretest.

when the test for loop completion occurs after the body of loop is executed, then it is called posttest.

Counter-controlled loops

A counting iterative control statement has a variable called the loop variable in which the count value is maintained.

It also includes some means of specifying the initial and terminal values of the loop variable and the difference between two sequential loop variable is called step size.

- The various design issues are:

1) What are the type and scope of the loop variable?
2) 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?
3) Should the loop parameters be evaluated only once, or once for every iteration?

Ex:- Ada's for statement

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

→
```
Count : Float := 1.35
for count in 1..10 loop
  Sum := Sum + count;
end loop;
```

C's for statement

```
for (expr1; expr2; expr3)
 loop body;
```

→
```
for (count = 1 ; count <=10; count++)
  {
    ----
  }
```

Logically controlled Loops

- In this the repetition control is based on a Boolean expression rather than a counter.
- Every counter-controlled loop can be implemented with logically controlled loops, but the reverse is not true.
- The design issues of logically controlled loops are:
  1) Should the control be pretest or post test
  2) Should the logically controlled loop be a special form of a counting loop or a separate statement.

Ex:

→ pretest loop

```
while (control_expression)
  loop body
```

→ post test loop

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

Unconditional Branching

An unconditional branch statement transfers execution control to a specified location in the program.

The unconditional branch (or) goto is the most powerful statement for controlling the flow of execution of program's statements.

- However using the goto carelessly can lead to serious problems.
- A few languages have been designed without a goto - for example Java, Ruby and Python.

Guarded commands

- Dijkstra's selection statement has the form

```
if  <Boolean expression> → <statement>
[]  <Boolean expression> → <statement>
    ⋮
[]  <Boolean expression> → <statement>
fi
```

- Each line in the selection statement consisting of a Boolean expression (a guard) and a statement or a statement sequence is called a guarded command.

- All of the Boolean expressions are evaluated each time the statement is reached during execution.
- If more than one expression is true, one of the corresponding statement can be chosen for execution.

Example

```
if i=0 → sum := sum+i
[] i>j → sum := sum+j
[] j>i → sum := sum+i.
```

- if i=0 and j>i, this statement choose one of the three assignment statements. otherwise a run time error occurs.