**UNIT -2**

**CHAPTER 1**

**DATA TYPES**

* 6.1 Introduction
* 6.2 Primitive Data Types
* 6.3 Character String Types
* 6.4 User-Defined Ordinal Types
* 6.5 Array Types
* 6.6 Associative Arrays
* 6.7 Record Types
* 6.8 Union Types
* 6.9 Pointer and Reference Types
* **6.1 Introduction:**
* A **data type** defines a collection of data values and a set of predefined operations on those values.
* Computer programs produce results by manipulating data.
* It is crucial that a language supports an appropriate collection of data types and structures.
* The contemporary concepts of data typing have evolved over the last 55 years.
* For example, in pre-90 FORTRANS, linked lists and binary trees were implemented with arrays.
* The data structures of COBOL took the first step away from the Fortran I model by allowing programmers to specify the accuracy of decimal data values.
* A better approach, introduced in ALGOL 68, is to provide a few basic types
* Clearly, this was one of the most important advances in the evolution of data type design.
* User-defined types also provide improved readability through the use of meaningful names for types.
* The two most common structured data types in the imperative languages are arrays and records, although the popularity of associative arrays has increased significantly in recent years.
* A **descriptor** is the collection of the attributes of a variable.
* In an implementation, a descriptor is an area of memory that stores the attributes of a variable.
* **6.2** **Primitive Data Types:**
* Data types that are not defined in terms of other types are called **primitive data types**.
* Nearly all programming languages provide a set of primitive datatypes.
* The three basic primitive data types are

1. Numeric Types,

2. Boolean Types

3. Character Types

* **6.2.1 Numeric Types:**
* Many early programming languages had only numeric primitive types.
* Numeric types still play a central role among the collections of types supported by contemporary languages.
* **6.2.1.1 Integer**
* The most common primitive numeric data type 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 languages, for example, C++ and C#, include unsigned integer types, which are

simply types for integer values without signs.

* Unsigned types are often used for binary data.
* A signed integer value is represented in a computer by a string of bits, with one of the bits representing the sign.
* Most integer types are supported directly by the hardware.
* One example of an integer type that is not supported directly by the hardware is the long integer type of Python
* Most computers now use a notation called **twos complement** to store negative integers, which is convenient for addition and subtraction.
* **6.2.1.2 Floating-Point:**
* **Floating**-**point** data types model real numbers, but the representations are only approximations for many real values.
* On most computers, floatingpoint numbers are stored in binary, which exacerbates the problem.
* Another problem with floating-point types is the loss of accuracy through arithmetic operations.
* Floating-point values are represented as fractions and exponents, a form that is borrowed from scientific notation.
* Most languages include two floating-point types, often called **float** and **double**.
* The float type is the standard size, usually being stored in four bytes of memory.
* The double type is provided for situations where larger fractional parts and/or a larger range of exponents is needed.
* The collection of values that can be represented by a floating-point type is defined in terms of precision and range.
* **Precision** is the accuracy of the fractional part of a value, measured as the number of bits.
* **Range** is a combination of the range of fractions and, more important, the range of exponents.
* **6.2.1.3 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)
* Languages that support a complex type include operations for arithmetic on complex values.
* **6.2.1.4 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# and F# also have 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.
* For example, the number 0.1 (in decimal) can be exactly represented in a decimal type, but not in a floating-point type.
* The disadvantages of decimal types are that the range of values is restricted because no exponents are allowed, and their representation in memory is mildly wasteful.
* Decimal types are stored very much like character strings, using binary codes for the decimal digits. These representations are called **binary coded** **decimal (BCD)**.
* **6.2.2 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, all operands with nonzero values are considered true, and zero is considered false.
* Although C99 and C++ have a Boolean type, they also allow numeric expressions to be used as if they were Boolean.
* This is not the case in the subsequent languages, Java and C#.
* Boolean types are often used to represent switches or flags in programs.
* **6.2.3 Character Types:**
* Character data are stored in computers as numeric codings.
* Traditionally, the most commonly used coding was the 8-bit code ASCII (American Standard Code for Information Interchange),
* This uses the values 0 to 127 to code 128 different characters.
* Because of the globalization of business and the need for computers to communicate with other computers around the world.
* The ASCII character set became inadequate. In response, in 1991, the Unicode Consortium published the UCS-2 standard, a 16-bit character set.
* This character code is often called Unicode. Unicode includes the characters from most of the world’s natural languages.
* **6.3 Character String Types:**
* A **character string type** is one in which the values consist of sequences of characters.
* Character strings a are an essential type for all programs that do character manipulation.
* **6.3.1 Design Issues**
* The two most important design issues that are specific to character string types are the following:
* 1.) Should strings be simply a special kind of character array or a primitive type?
* 2.)Should strings have static or dynamic length?
* **6.3.2 Strings and Their Operations**
* The most common string operations are assignment, catenation, substring reference, comparison, and pattern matching.
* A **substring reference** is a reference to a substring of a given string.
* Substring references are discussed in the more general context of arrays, where the substring references are called **slices**.
* In general, both assignment and comparison operations on character strings are complicated by the possibility of string operands of different lengths.
* For example, what happens when a longer string is assigned to a shorter string, or vice versa?
* Pattern matching is another fundamental character string operation. In some languages, pattern matching is supported directly in the language.
* In others, it is provided by a function or class library.
* 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.
* This is the approach taken by C and C++.
* C and C++ use **char** arrays to store character strings. These languages provide a collection of string operations through standard libraries.
* The library operations simply carry out their operations until the null character appears in the string being operated on.
* For example, consider the following declaration:
* **char** str[] = "apples";
* In this example, str is an array of **char** elements, specifically apples0, where 0 is the null character.
* Some of the most commonly used library functions for character strings in C and C++ are strcpy, which moves strings;

strcat, which catenates one given string onto another;

strcmp, which lexicographically compares two given strings; and

strlen, which returns the number of characters, not counting the null, in the given string.

* The string manipulation functions of the C standard library, which are also available in C++, are inherently unsafe and have led to numerous programming errors.
* The problem is that the functions in this library that move string data do not guard against overflowing the destination.
* For example, consider the following call to strcpy:

strcpy(dest, src);

* If the length of dest is 20 and the length of src is 50, strcpy will write over the 30 bytes that follow dest.
* The point is that strcpy does not know the length of dest, so it cannot ensure that the memory following it will not be overwritten.
* The same problem can occur with several of the other functions in the C string library.
* In Java, strings are supported by the String class, whose values are constant strings, and the StringBuffer class,
* Python includes strings as a primitive type and has operations for substring reference, catenation, indexing to access individual characters, as well as methods for searching and replacement.
* Perl, JavaScript, Ruby, and PHP include built-in pattern-matching operations.
* In these languages, the pattern-matching expressions are somewhat loosely based on mathematical regular expressions. In fact, they are often called **regular expressions**.
* **6.3.3 String Length Options**
* There are several design choices regarding the length of string values.
* First, the length can be static and set when the string is created. Such a string is called a **static length string**.
* This is 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# and F#.
* The second option is to allow strings to have varying length up to a declared and fixed maximum set by the variable’s definition, as exemplified by the strings in C and the C-style strings of C++.
* These are called **limited** **dynamic length strings**. Such string variables can store any number of characters between zero and the maximum.
* The third option is to allow strings to have varying length with no maximum, as in JavaScript, Perl, and the standard C++ library.
* These are called **dynamic length strings**. This option requires the overhead of dynamic storage allocation and deallocation but provides maximum flexibility.
* Ada 95+ supports all three string length options.
* **6.3.4 Evaluation**
* String types are important to the writability of a language.
* Dealing with strings as arrays can be more cumbersome than dealing with a primitive string type.
* Then, a simple assignment of one string to another would require a loop.
* String operations such as simple pattern matching and catenation are essential and should be included for string type values.
* Although dynamic length strings are obviously the most flexible, the overhead of their implementation must be weighed against that additional flexibility.
* **6.3.5 Implementation of Character String Types**
* Character string types could be supported directly in hardware; but in most cases, software is used to implement string storage, retrieval, and manipulation.
* When character string types are represented as character arrays, the language often supplies few operations.
* A descriptor for a static character string type, which is required only during compilation, has three fields.
* The first field of every descriptor is the name of the type. In the case of static character strings,
* The second field is the type’s length (in characters).
* The third field is the address of the first character.
* This descriptor is shown in Figure 6.2.
* Limited dynamic strings require a run-time descriptor to store both the fixed maximum length and the current length, as shown in Figure 6.3.
* Dynamic length strings require a simpler run-time descriptor because only the current length needs to be stored.



* The limited dynamic strings of C and C++ do not require run-time descriptors, because the end of a string is marked with the null character.
* Static length and limited dynamic length strings require no special dynamic storage allocation.
* Dynamic length strings require more complex storage management.
* The length of a string, and therefore the storage to which it is bound, must grow and shrink dynamically.
* There are three approaches to supporting the dynamic allocation and deallocation that is required for dynamic length strings.
* First, strings can be stored in a linked list, so that when a string grows, the newly required cells can come from anywhere in the heap.
* The drawbacks to this method are the extra storage occupied by the links in the list representation and the necessary complexity of string operations.
* The second approach is to store strings as arrays of pointers to individual characters allocated in the heap.
* This method still uses extra memory, but string processing can be faster than with the linked-list approach.
* The third alternative is to store complete strings in adjacent storage cells.
* The problem with this method arises when a string grows:
* How can storage that is adjacent to the existing cells continue to be allocated for the string variable?
* Frequently, such storage is not available. Instead, a new area of memory is found that can store the complete new string, and the old part is moved to this area. Then, the memory cells used for the old string are deallocated.
* **6.4 User-Defined Ordinal Types:**
* An **ordinal type** is one in which the range of possible values can be easily associated with the set of positive integers.
* In Java, for example, the primitive ordinal types are **integer**, **char**, and **boolean**.
* There are two user-defined ordinal types that have been supported by programming languages:

Enumeration and

subrange.

* **6.4.1 Enumeration Types:**
* An **enumeration type** is one in which all of the possible values, which are named constants, are provided, or enumerated, in the definition.
* Enumeration types provide a way of defining and grouping collections of named constants,

which are called **enumeration constants**.

* The definition of a typical enumeration type is shown in the following C# example:

**enum** days {Mon, Tue, Wed, Thu, Fri, Sat, Sun};

* The design issues for enumeration types are as follows:
* 1.)Is an enumeration constant allowed to appear in more than one type definition, and if so, how is the type of an occurrence of that constant in the program checked?
* 2.)Are enumeration values coerced to integer?
* 3.) Are any other types coerced to an enumeration type?
* If an **int** type value is coerced to an enumeration type, then an enumeration type variable could be assigned any integer value, whether it represented an enumeration constant or not.
* **6.4.1.1 Designs**
* In languages that do not have enumeration types, programmers usually simulate them with integer values.
* For example, suppose we needed to represent colors in a C program and C did not have an enumeration type.
* We might use 0 to represent blue, 1 to represent red, and so forth. These values could be
* defined as follows:

**int** red = 0, blue = 1;

* Now, in the program, we could use red and blue as if they were of a color type.
* C and Pascal were the first widely used languages to include an enumeration data type. C++ includes C’s enumeration types.
* In C++, we could have the following:

**enum** colors {red, blue, green, yellow, black};

colors myColor = blue, yourColor = red;

* The colors type uses the default internal values for the enumeration constants, 0, 1, . . . , although the constants could have been assigned any integer literal .
* The enumeration values are coerced to **int** when they are put in integer context.
* For example, if the current value of myColor is blue, then the expression

myColor++

* would assign green to myColor.
* C++ also allows enumeration constants to be assigned to variables of any numeric type, though that would likely be an error.
* However, no other type value is coerced to an enumeration type in C++.
* For example,

myColor = 4;

* is illegal in C++.
* This assignment would be legal if the right side had been cast to colors type.
* This prevents some potential errors.
* **6.4.1.2 Evaluation**
* Enumeration types can provide advantages in both readability and reliability.
* Readability is enhanced very directly: Named values are easily recognized, whereas coded values are not.
* In the area of reliability, the enumeration types of Ada, C#, F#, and Java 5.0 provide two advantages:
* (1) No arithmetic operations are legal on enumeration types; this prevents adding days of the week, for example, and
* (2) second, no enumeration variable can be assigned a value outside its defined range.
* If the colors enumeration type has 10 enumeration constants and uses 0..9 as its internal values, no number greater than 9 can be assigned to a colors type variable.
* **6.4.2 Subrange Types**
* A **subrange type** is a contiguous subsequence of an ordinal type.
* For example, 12..14 is a subrange of integer type. Subrange types were introduced by Pascal and are included in Ada.
* There are no design issues that are specific to subrange types.
* **6.4.2.1 Ada’s Design**
* In Ada, subranges are included in the category of types called subtypes.
* For example, consider the following declarations:

**type** Days **is** (Mon, Tue, Wed, Thu, Fri, Sat, Sun);

**subtype** Weekdays is Days **range** Mon..Fri;

**subtype** Index **is** Integer **range** 1..100;

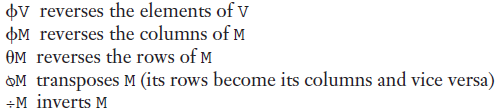
* In these examples, the restriction on the existing types is in the range of possible values. All of the operations defined for the parent type are also defined example, in

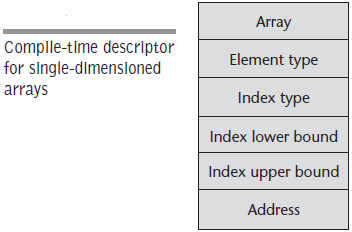
Day1 : Days;

Day2 : Weekdays;

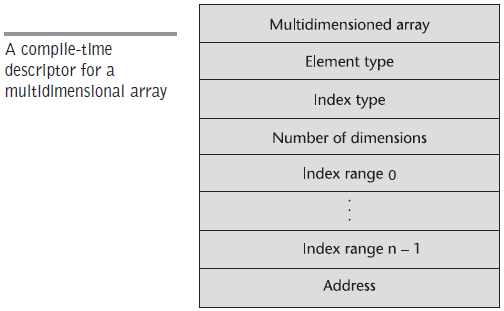
. .

Day2 := Day1;

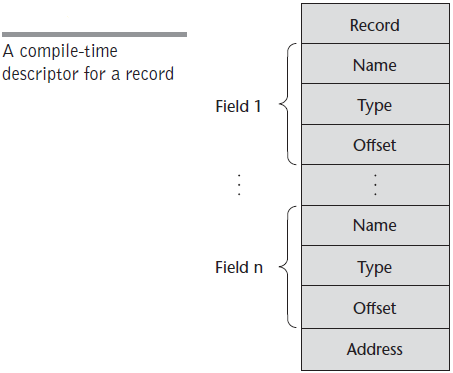
* The assignment is legal unless the value of Day1 is Sat or Sun. The compiler must generate range-checking code for every assignment to a subrange variable.
* While types are checked for compatibility at compile time, subranges require run-time range checking.
* One of the most common uses of user-defined ordinal types is for the indices of arrays.
* They can also be used for loop variables. In fact, subranges of ordinal types are the only way the range of Ada **for** loop variables can be specified.
* **6.4.2.2 Evaluation**
* Subrange types enhance readability by making it clear to readers that variables of subtypes can store only certain ranges of values.
* Reliability is increased with subrange types, because assigning a value to a subrange variable that is outside the specified range is detected as an error, either by the compiler or by the run-time system
* Ada has subrange types.
* **6.4.3 Implementation of User-Defined Ordinal Types**
* As discussed earlier, enumeration types are usually implemented as integers.
* Without restrictions on ranges of values and operations, this provides no increase in reliability.
* Subrange types are implemented in exactly the same way as their parent types, except that range checks must be implicitly included by the compiler in every assignment of a variable or expression to a subrange variable.
* This step increases code size and execution time, but is usually considered well worth the cost.
* Also, a good optimizing compiler can optimize away some of the checking.
* **6.5 Array Types:**
* An **array** is a homogeneous aggregate of data elements in which an individual element is identified by its position in the aggregate, relative to the first element. The individual data elements of an array are of the same type.
* 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. In these languages, pointers and references are restricted to point to or reference a single type.
* In some other languages, such as JavaScript, Python, and Ruby, variables are typeless references
* to objects or data values.
* **6.5.1. Design Issues:** The primary design issues specific to arrays are the following:
* • 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 arrays be initialized when they have their storage allocated?
* • What kinds of slices are allowed, if any?
* **6.5.2. Arrays and Indices:**
* Specific elements of an array are referenced by means of a two-level syntactic mechanism, where the first part is the aggregate name, and the second part is a possibly dynamic selector consisting of one or more items known as **subscripts** or **indices**. If all of the subscripts in a reference are constants, the selector is static; otherwise, it is dynamic. The selection operation can be thought of as a mapping from the array name and the set of subscript values to an element in the aggregate. Indeed, arrays are sometimes called **finite mappings**. Symbolically, this mapping can be shown as
* array\_name(subscript\_value\_list) → element
* The array name is followed by the list of subscripts, which is surrounded by either parentheses or brackets. In some languages that provide multidimensioned arrays as arrays of arrays, each subscript appears in its own brackets. A problem with using parentheses to enclose subscript expressions is that they often are also used to enclose the parameters in subprogram calls; this use makes references to arrays appear exactly like those calls. For example, consider the following Ada assignment statement:
* Sum := Sum + B(I);
* Because parentheses are used for both subprogram parameters and array subscripts in Ada, both program readers and compilers are forced to use other information to determine whether B(I) in this assignment is a function call or a reference to an array element.
* The designers of Ada specifically chose parentheses to enclose subscripts so there would be uniformity between array references and function calls in expressions. Array element references map the subscripts to a particular element of the array. Function calls map the actual parameters to the function
* definition and, eventually, a functional value.
* Many contemporary languages do not specify range checking of subscripts, but Java, ML, and C# do. By default, Ada checks the range of all subscripts, but this feature can be disabled by the programmer.
* Subscripting in Perl is a bit unusual in that although the names of all arrays begin with at signs (@), because array elements are always scalars and the names of scalars always begin with dollar signs ($), references to array elements use dollar signs rather than at signs in their names. For example, for the array @list, the second element is referenced with $list[1].
* **6.5.3. 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.
* 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; in Fortran 95+ it defaults to 1 but can be set to any integer literal. In some other languages, the lower bounds of the subscript ranges must be specified by the programmer.
* There are five categories of arrays, based on the binding to subscript ranges, the binding to storage, and from where the storage is allocated. In the first four of these categories, once the subscript ranges are bound and the storage is allocated, they remain fixed for the lifetime of the variable.
* A **static array** is one in which the subscript ranges are statically bound and storage allocation is static (done before run time). The advantage of static arrays is efficiency: No dynamic allocation or deallocation is required. The disadvantage is that the storage for the array is fixed for the entire execution
* time of the program.
* A **fixed stack-dynamic array** is one in which the subscript ranges are statically bound, but the allocation is done at declaration elaboration time during execution. The advantage of fixed stack-dynamic arrays over static arrays is space efficiency. The disadvantage is the required allocation and deallocation time.
* A **stack-dynamic array** is one in which both the subscript ranges and the storage allocation are dynamically bound at elaboration time. Once the subscript ranges are bound and the storage is allocated, however, they remain fixed during the lifetime of the variable. The advantage of stack-dynamic arrays over static and fixed stack-dynamic arrays is flexibility.
* A **fixed heap-dynamic array** is similar to a fixed stack-dynamic array, in that the subscript ranges and the storage binding are both fixed after storage is allocated. The 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. The advantage of fixed heap-dynamic
* arrays is flexibility—the array’s size always fits the problem. The disadvantage is allocation time from the heap, which is longer than allocation time from the stack.
* A **heap-dynamic array** is one in which the binding of subscript ranges and storage allocation is dynamic and can change any number of times during the array’s lifetime. The advantage of heap-dynamic arrays over the others is flexibility: Arrays can grow and shrink during program execution as the need for
* space changes. The disadvantage is that allocation and deallocation take longer and may happen many times during execution of the program.
* Arrays declared in C and C++ functions that include the **static** modifier are static. 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.
* In Java, all non-generic arrays are fixed heap-dynamic. Once created, these arrays keep the same subscript ranges and storage. C# also provides the same kind of arrays.
* Perl,Javascript,Ruby and Python supports heap-dynamic arrays.
* **6.5.4. Array Initialization:**
* 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. An array aggregate for a single-dimensioned array is a list of literals delimited by parentheses and slashes. For example, we could have
* **Integer**, Dimension (3) :: List = (/0, 5, 5/)
* C, C++, Java, and C# also allow initialization of their arrays, but with one new twist: In the C declaration the compiler sets the length of the array.
* **int** list [] = {4, 5, 7, 83};
* Character strings in C and C++ are implemented as arrays of **char**. These arrays can be initialized to string constants, as in
* **char** name [] = "freddie";
* The array name will have eight elements, because all strings are terminated with a null character (zero), which is implicitly supplied by the system for string constants.
* Arrays of strings in C and C++ can also be initialized with string literals. In this case, the array is one of pointers to characters. For example,
* **char** \*names [] = {"Bob", "Jake", "Darcie"};
* This example illustrates the nature of character literals in C and C++. In the previous example of a string literal being used to initialize the **char** array name, the literal is taken to be a **char** array. But in the latter example (names), the literals are taken to be pointers to characters, so the array is an array of pointers to characters. For example, names[0] is a pointer to the letter 'B' in the literal character array that contains the characters 'B', 'o', 'b', and the null character.
* In Java, similar syntax is used to define and initialize an array of references to String objects. For example,
* String[] names = ["Bob", "Jake", "Darcie"];
* Ada provides two mechanisms for initializing arrays in the declaration statement: by listing them in the order in which they are to be stored, or by directly assigning them to an index position using the => operator, which in Ada is called an **arrow**. For example, consider the following:
* List : **array** (1..5) **of** Integer := (1, 3, 5, 7, 9);
* Bunch : **array** (1..5) **of** Integer := (1 => 17, 3 => 34,**others** => 0);
* In the first statement, all the elements of the array List have initializing values, which are assigned to the array element locations in the order in which they appear. In the second, the first and third array elements are initialized using direct assignment, and the **others** clause is used to initialize the remaining elements.
* **6.5.5. Array Operations:**
* An array operation is one that operates on an array as a unit. The most common
* array operations are assignment, catenation, comparison for equality and inequality, and slices.
* The C-based languages do not provide any array operations, except through the methods of Java, C++, and C#. Perl supports array assignments but does not support comparisons.
* Ada allows array assignments, including those where the right side is an aggregate value rather than an array name. Ada also provides catenation, specified by the ampersand (&).
* Python’s arrays are called lists, although they have all the characteristics of dynamic arrays. Because the objects can be of any types, these arrays are heterogeneous. Python provides array assignment, although it is only a reference change. Python also has operations for array catenation (+) and element membership (**in**).
* Like Python, the elements of Ruby’s arrays are references to objects. And like Python, when a == operator is used between two arrays, the result is true only if the two arrays have the same length and the corresponding elements are equal.
* Fortran 95+ includes a number of array operations that are called **elemental** because they are operations between pairs of array elements. For example, the add operator (+) between two arrays results in an array of the sums of the element pairs of the two arrays.
* Arrays and their operations are the heart of APL; it is the most powerful array-processing language ever devised. Because of its relative obscurity and its lack of effect on subsequent languages, however, we present here only a glimpse into its array operations.
* In APL, the four basic arithmetic operations are defined for vectors (single-dimensioned arrays) and matrices, as well as scalar operands. For example,
* A + B
* is a valid expression, whether A and B are scalar variables, vectors, or matrices.
* APL includes a collection of unary operators for vectors and matrices,
* some of which are as follows (where V is a vector and M is a matrix):
* 
* **6.5.6. 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. Rectangular arrays model rectangular tables exactly.
* A **jagged array** is one in which the lengths of the rows need not be the same. For example, a jagged matrix may consist of three rows, one with 5 elements, one with 7 elements, and one with 12 elements. This also applies to the columns and higher dimensions. So, if there is a third dimension (layers), each layer can have a different number of elements.
* C, C++, and Java support jagged arrays but not rectangular arrays. In those languages, a reference to an element of a multidimensioned array uses a separate pair of brackets for each dimension. For example,
* myArray[3][7]
* Fortran, Ada, C#, and F# support rectangular arrays. (C# and F# also support jagged arrays.) In these cases, all subscript expressions in references to elements are placed in a single pair of brackets. For example,
* myArray[3, 7]
* **6.5.7. Slices:**
* A **slice** of an array is some substructure of that array. For example, if A is a matrix, then the first row of A is one possible slice, as are the last row and the first column. It is important to realize that a slice is not a new data type.
* Consider the following Python declarations:
* vector = [2, 4, 6, 8, 10, 12, 14, 16]
* mat = [[1, 2, 3],[4, 5, 6],[7, 8, 9]]
* The syntax of a Python slice reference is a pair of numeric expressions separated by a colon. The
* first is the first subscript of the slice; the second is the first subscript after the last subscript in the slice. Therefore, vector[3:6] is a three-element array with the fourth through sixth elements of vector (those elements with the subscripts 3, 4, and 5). A row of a matrix is specified by giving just one subscript. For example, mat[1] refers to the second row of mat; a part of a row can be specified with the same syntax as a part of a single dimensioned array. For example, mat[0][0:2] refers to the first and second element of the first row of mat, which is [1, 2].
* Python also supports more complex slices of arrays. For example, vector[ 0:7:2] references every other element of vector, up to but not including the element with the subscript 7, starting with the subscript 0, which is [2, 6, 10, 14].
* Perl supports slices of two forms, a list of specific subscripts or a range of subscripts. For example,
* @list[1..5] = @list2[3, 5, 7, 9, 13];
* Ruby supports slices with the slice method of its Array object, which can take three forms of parameters. A single integer expression parameter is interpreted as a subscript, in which case slice returns the element with the given subscript. If slice is given two integer expression parameters, the first is
* interpreted as a beginning subscript and the second is interpreted as the number of elements in the slice.
* **6.5.8. Evaluation:**
* Arrays have been included in virtually all programming languages. The primary advances since their introduction in Fortran I have been the inclusion of all ordinal types as possible subscript types, slices, and, of course, dynamic arrays.
* **6.5.9. Implementation of Array Types:**
* Implementing arrays requires considerably more compile-time effort than does implementing primitive types. The code to allow accessing of array elements must be generated at compile time. At run time, this code must be executed to produce element addresses. There is no way to precompute the address to be accessed by a reference such as
* list[k]
* A single-dimensioned array is implemented as a list of adjacent memory cells. Suppose the array list is defined to have a subscript range lower bound of 0. The access function for list is often of the form
* address(list[k]) = address(list[0]) + k \* element\_size
* where the first operand of the addition is the constant part of the access function, and the second is the variable part.
* If the element type is statically bound and the array is statically bound to storage, then the value of the constant part can be computed before run time. However, the addition and multiplication operations must be done at run time. The generalization of this access function for an arbitrary lower bound is
* address(list[k]) = address(list[lower\_bound]) + ((k - lower\_bound) \* element\_size)
* The compile-time descriptor for single-dimensioned arrays can have the form shown in Figure. The descriptor includes information required to construct the access function. If run-time checking of index ranges is not done and the attributes are all static, then only the access function is required during
* execution; no descriptor is needed. If run-time checking of index ranges is done, then those index ranges may need to be stored in a run-time descriptor. If the subscript ranges of a particular array type are static, then the ranges may be incorporated into the code that does the checking, thus eliminating the need for
* the run-time descriptor. If any of the descriptor entries are dynamically bound, then those parts of the descriptor must be maintained at run time.



* The multidimensional arrays, that is, those that are not arrays of arrays, are more complex to implement than single-dimensioned arrays, although the extension to more dimensions is straightforward. There are two ways in which multidimensional arrays can be mapped to one dimension: row major order and column major order. In **row major order**, the elements of the array that have as their first subscript the lower bound value of that subscript are stored first, followed by the elements of the second value of
* the first subscript, and so forth. If the array is a matrix, it is stored by rows. For example, if the matrix had the values
* 3 4 7
* 6 2 5
* 1 3 8
* it would be stored in row major order as 3, 4, 7, 6, 2, 5, 1, 3, 8
* In **column major order**, the elements of an array that have as their last subscript the lower bound value of that subscript are stored first, followed by the elements of the second value of the last subscript, and so forth. If the array is a matrix, it is stored by columns. If the example matrix were stored in column
* major order, it would have the following order in memory: 3, 6, 1, 4, 2, 3, 7, 5, 8
* For each dimension of an array, one add and one multiply instruction are required for the access function. Therefore, accesses to elements of arrays with several subscripts are costly. The compile-time descriptor for a multidimensional array is shown in Figure.

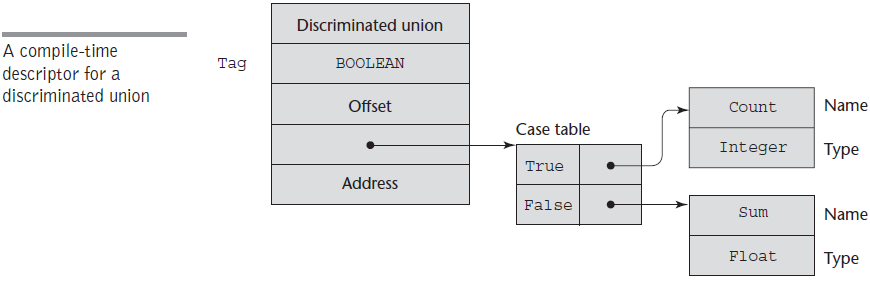


* **6.6 Associative Arrays:-**
* An **associative array** is an unordered collection of data elements that are indexed by an equal number of values called **keys**.
* **6.6.1. Structure and Operations:**
* In Perl, associative arrays are called **hashes**, because in the implementation their elements are stored and retrieved with hash functions. The namespace for Perl hashes is distinct: Every hash variable name must begin with a percent sign (%). Each hash element consists of two parts: a key, which is a string, and a value, which is a scalar (number, string, or reference). Hashes can be set to literal values with the assignment statement, as in
* %salaries = ("Gary" => 75000, "Perry" => 57000,
* "Mary" => 55750, "Cedric" => 47850);
* Individual element values are referenced using notation that is similar to that used for Perl arrays. The key value is placed in braces and the hash name is replaced by a scalar variable name that is the same except for the first character. For example,
* $salaries{"Perry"} = 58850;
* A new element is added using the same assignment statement form. An element can be removed from the hash with the **delete** operator, as in
* **delete** $salaries{"Gary"};
* The entire hash can be emptied by assigning the empty literal to it, as in
* @salaries = ();
* The exists operator returns true or false, depending on whether its operand key is an element in the hash. For example,
* **if** (exists $salaries{"Shelly"}) . . .
* Python’s associative arrays, which are called **dictionaries**, are similar to those of Perl, except the values are all references to objects. The associative arrays supported by Ruby are similar to those of Python, except that the keys can be any object,6 rather than just strings. There is a progression from Perl’s hashes, in which the keys must be strings, to PHP’s arrays, in which the keys can be integers or strings, to Ruby’s hashes, in which any type object can be a key.
* **6.6.2. 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. The elements in PHP’s arrays are placed in memory through a hash function.
* **6.7 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. For example, information about a college student might include name, student number, grade point average, and so forth. A data type for such a collection might use a character string for the name, an integer for the student number, a floating point for the grade point average, and so forth. Records are designed for this kind of need.
* The following design issues are specific to records:
* • What is the syntactic form of references to fields?
* • Are elliptical references allowed?
* **6.7.1. Definition of Records:**
* The fundamental difference between a record and an array is that record elements, or **fields**, are not referenced by indices. 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.
* Ada uses a different syntax for records; rather than using the level numbers of COBOL, record structures are indicated in an orthogonal way by simply nesting record declarations inside record declarations. 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;
* **6.7.2. References to Record Fields:**
* References to the individual fields of records are syntactically specified by several different methods, two of which name the desired field and its enclosing records. COBOL field references have the form
* field\_name OF record\_name\_1 OF **. . .** OF record\_name\_n
* where the first record named is the smallest or innermost record that contains the field. The next record name in the sequence is that of the record that contains the previous record, and so forth. For example, the MIDDLE field in the COBOL record example above can be referenced with
* MIDDLE OF EMPLOYEE-NAME OF EMPLOYEE-RECORD
* Most of the other languages use **dot notation** for field references, where the components of the reference are connected with periods. Names in dot notation have the opposite order of COBOL references: They use the name of the largest enclosing record first and the field name last. For example, the following is a reference to the field Middle in the earlier Ada record example:
* Employee\_Record.Employee\_Name.Middle
* C and C++ use this same syntax for referencing the members of their structures.
* 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 named in the reference. Both the COBOL and the Ada example field references above are fully qualified.
* COBOL allows **elliptical references** to record fields. In an elliptical reference, the field is named, but any or all of the enclosing record names can be omitted, as long as the resulting reference is unambiguous in the referencing environment. For example, FIRST, FIRST OF EMPLOYEE-NAME, and FIRST OF EMPLOYEE-RECORD are elliptical references to the employee’s first name in the COBOL record declared above.
* **6.7.2. Evaluation:**
* Records are used when the collection of data values is heterogeneous and the different fields are not processed in the same way. Also, the fields of a record often need not be processed in a particular order. Field names are like literal, or constant, subscripts.
* **6.7.3. Implementation of Record Types:**
* The fields of records are stored in adjacent memory locations. But because the sizes of the fields are not necessarily the same, the access method used for arrays is not used for records. Instead, the offset address, relative to the beginning of the record, is associated with each field. Field accesses are all handled using these offsets. The compile-time descriptor for a record has the general form shown in Figure.



* **6.8. Union Types:-**
* A **union** is a type whose variables may store different type values at different times during program execution. 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.
* **6.8.1. Design Issues:** The primary design issues that are particular to union types are the following:
* • Should type checking be required? Note that any such type checking must be dynamic.
* • Should unions be embedded in records?
* **6.8.2. Discriminated Versus Free Unions:** C and C++ provide union constructs in which there is no language support for type checking. In C and C++, the **union** construct is used to specify union structures. The unions in these languages are called **free unions**, because programmers are allowed complete freedom from type checking in their use. For example, consider the following C union:
* **union** flexType {
* **int** intEl;
* **float** floatEl;
* };
* **union** flexType el1;
* **float** x;
* . . .
* el1.intEl = 27;
* x = el1.floatEl;
* This last assignment is not type checked, because the system cannot determine the current type of the current value of el1, so it assigns the bit string representation of 27 to the **float** variable x, which of course is nonsense.
* Type checking of unions requires that each union construct include a type indicator. Such an indicator is called a **tag**, or **discriminant**, and a union with a discriminant is called a **discriminated union**. The first language to provide discriminated unions was ALGOL 68. They are now supported by Ada.
* **6.8.3. Ada Union Types:**
* The Ada design for discriminated unions, which is based on that of its predecessor
* language, Pascal, allows the user to specify variables of a variant record type that will store only one of the possible type values in the variant. In this way, the user can tell the system when the type checking can be static. Such a restricted variable is called a **constrained variant variable**.
* 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**;
* The following two statements declare variables of type Figure:
* Figure\_1 : Figure;
* Figure\_2 : Figure(Form => Triangle);
* Figure\_1 is declared to be an unconstrained variant record that has no initial value. Its type can change by assignment of a whole record, including the discriminant, as in the following:
* Figure\_1 := (Filled => True,
* Color => Blue,
* Form => Rectangle,
* Side\_1 => 12,
* Side\_2 => 3);
* The variable Figure\_2 is declared constrained to be a triangle and cannot be changed to another variant.
* This form of discriminated union is safe, because it always allows type checking, although the references to fields in unconstrained variants must be dynamically checked.

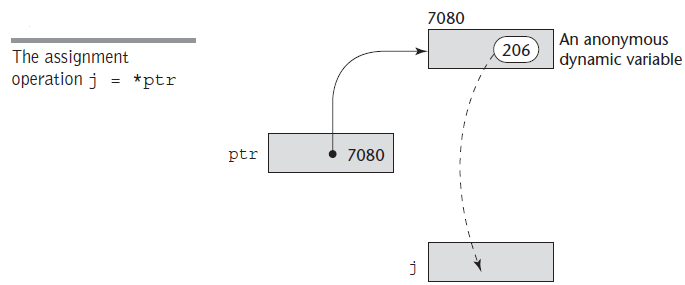
* **6.8.4. Evaluation:**
* Unions are potentially unsafe constructs in some languages. They are one of the reasons why C and C++ are not strongly typed: These languages do not allow type checking of references to their unions. On the other hand, unions can be safely used, as in their design in Ada, ML, Haskell.
* **6.8.5. Implementation of Union Types:**
* Unions are implemented by simply using the same address for every possible variant. Sufficient storage for the largest variant is allocated. The tag of a discriminated
* union is stored with the variant in a record like structure.
* At compile time, the complete description of each variant must be stored. This can be done by associating a case table with the tag entry in the descriptor. The case table has an entry for each variant, which points to a descriptor for that particular variant. To illustrate this arrangement, consider the following Ada example:
* **type** Node (Tag : Boolean) **is**
* **record**
* **case** Tag **is**
* **when** True => Count : Integer;
* **when** False => Sum : Float;
* **end case**;
* **end record**;
* The descriptor for this type could have the form shown in Figure.



* **6.9 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 are designed for two distinct kinds of uses. First, pointers provide some of the power of indirect addressing, which is frequently used in assembly language programming. Second, pointers provide a way to manage dynamic storage. A pointer can be used to access a location in an area where storage is dynamically allocated called a **heap**.
* Variables that are dynamically allocated from the heap are called **heap dynamic variables**. Variables without names are called **anonymous variables**.
* **6.9.1. Design Issues:**

The primary design issues particular to pointers are the following:

* • What are the scope and lifetime of a pointer variable?
* • What is the lifetime of a heap-dynamic variable (the value a pointer references)?
* • Are pointers restricted as to the type of value to which they can point?
* • Are pointers used for dynamic storage management, indirect addressing, or both?
* • Should the language support pointer types, reference types, or both?
* **6.9.2. Pointer Operations:**
* Languages that provide a pointer type usually include two fundamental 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, then the allocation mechanism, whether by operator or built-in subprogram, serves to initialize the pointer variable. If pointers are used for indirect addressing to variables that are not heap dynamic, then there must be an explicit operator or built-in subprogram for fetching the address of a variable, which can then be assigned to the pointer variable.
* An occurrence of 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. However, 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. In this case, 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.
* Dereferencing of pointers can be either explicit or implicit. In Fortran 95+ it is implicit, but in many other contemporary languages, it occurs only when explicitly specified. In C++, it is explicitly specified with the asterisk (\*) as a prefix unary operator. Consider the following example of dereferencing: 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 in Figure.



* When pointers point to records, the syntax of the references to the fields of these records varies among languages. In C and C++, there are two ways a pointer to a record can be used to reference a field in that record. If a pointer variable p points to a record with a field named age, (\*p).age can be used to
* refer to that field. The operator ->, when used between a pointer to a record and a field of that record, combines dereferencing and field reference. For example, the expression p -> age is equivalent to (\*p).age. In Ada, p.age can be used, because such uses of pointers are implicitly dereferenced.
* **6.9.3. Pointer Problems:**
* The first high-level programming language to include pointer variables was
* PL/I, in which pointers could be used to refer to both heap-dynamic variables and other program variables. The pointers of PL/I were highly flexible, but their use could lead to several kinds of programming errors. Some recent languages, such as Java, have replaced pointers completely with reference types, which, along with implicit deallocation, minimize the primary problems with pointers.
* **6.9.3.1. 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.
* The following sequence of operations creates a dangling pointer in many languages:
* 1. A new heap-dynamic variable is created and pointer p1 is set to point at it.
* 2. Pointer p2 is assigned p1’s value.
* 3. The heap-dynamic variable pointed to by p1 is explicitly deallocated (possibly setting p1 to nil), but p2 is not changed by the operation. p2 is now a dangling pointer. If the deallocation operation did not change
* p1, both p1 and p2 would be dangling. (Of course, this is a problem of aliasing—p1 and p2 are aliases.)
* For example, in C++ we could have the following:
* **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.
* **6.9.3.2. Lost Heap-Dynamic Variables:**
* A **lost heap-dynamic variable** is an allocated heap-dynamic variable that is no longer accessible to the user program. Such variables are often called **garbage**, because they are not useful for their original purpose, and they also cannot be reallocated for some new use in the program. Lost heap-dynamic variables are most often created by the following sequence of operations:
* 1. Pointer p1 is set to point to a newly created heap-dynamic variable.
* 2. p1 is later set to point to another newly created heap-dynamic variable.
* The first heap-dynamic variable is now inaccessible, or lost. This is sometimes called **memory leakage**. Memory leakage is a problem, regardless of whether the language uses implicit or explicit deallocation.
* **6.9.4. Pointers in Ada:**
* Ada’s pointers are called **access** types. The dangling-pointer problem is partially alleviated by Ada’s design, at least in theory. A heap-dynamic variable may be (at the implementor’s option) implicitly deallocated at the end of the scope of its pointer type; thus, dramatically lessening the need for explicit deallocation. Because heap-dynamic variables can be accessed by variables of only one type, when the end of the scope of that type declaration is reached, no pointers can be left pointing at the dynamic variable. This diminishes the problem, because improperly implemented explicit deallocation is the major source of dangling pointers.
* Unfortunately, the Ada language also has an explicit deallocator, Unchecked\_Deallocation. Its name is meant to discourage its use, or at least warn the user of its potential problems. Unchecked\_Deallocation can cause dangling pointers.
* **6.9.5. Pointers in C and C++:**
* C and C++ pointers can point at any variable, regardless of where it is allocated. In fact, they can point anywhere in memory, whether there is a variable there or not, which is one of the dangers of such pointers. In C and C++, the asterisk (\*) denotes the dereferencing operation, and the ampersand (&) denotes the operator for producing the address of a variable. For example, consider the following code:
* **int** \*ptr;
* **int** count, init;
* . . .
* ptr = &init;
* count = \*ptr;
* The assignment to the variable ptr sets it to the address of init. The assignment to count dereferences ptr to produce the value at init, which is then assigned to count. So, the effect of the two assignment statements is to assign the value of init to count.
* Pointer arithmetic is also possible in some restricted forms. For example, if ptr is a pointer variable that is declared to point at some variable of some data type, then
* ptr + index
* is a legal expression. The semantics of such an expression is as follows. Instead of simply adding the value of index to ptr, the value of index is first scaled by the size of the memory cell (in memory units) to which ptr is pointing (its base type). For example, if ptr points to a memory cell for a type that is four memory units in size, then index is multiplied by 4, and the result is added to ptr.
* In C and C++, all arrays use zero as the lower bound of their subscript ranges, and array names without subscripts always refer to the address of the first element. Consider the following declarations:
* **int** list [10];
* **int** \*ptr;
* Consider the assignment
* ptr = list;
* which assigns the address of list[0] to ptr, because an array name without a subscript is interpreted as the base address of the array. Given this assignment, the following are true:
* • \*(ptr + 1) is equivalent to list[1].
* • \*(ptr + index) is equivalent to list[index].
* • ptr[index] is equivalent to list[index].
* **6.9.6. Reference Types:**
* A reference type variable is similar to a pointer, with one important and fundamental difference: A pointer refers to an address in memory, while a reference refers to an object or a value in memory.
* C++ includes a special kind of reference type that is used primarily for the formal parameters in function definitions. A C++ reference type variable is a constant pointer that is always implicitly dereferenced.
* Reference type variables are specified in definitions by preceding their names with ampersands (&). For example,
* int result = 0;
* int &ref\_result = result;
* . . .
* ref\_result = 100;
* When used as formal parameters in function definitions, C++ reference types provide for two-way communication between the caller function and the called function. This is not possible with nonpointer primitive parameter types, because C++ parameters are passed by value. Passing a pointer
* as a parameter accomplishes the same two-way communication, but pointer formal parameters require explicit dereferencing, making the code less readable and less safe. Reference parameters are referenced in the called function exactly as are other parameters.
* Unlike C++ reference variables, Java reference variables can be assigned to refer to different class instances; they are not constants. All Java class instances are referenced by reference variables. That is, in fact, the only use of reference variables in Java.
* In the following, String is a standard Java class:
* String str1;
* . . .
* str1 = "This is a Java literal string";
* In this code, str1 is defined to be a reference to a String class instance or object. It is initially set to null. The subsequent assignment sets str1 to reference the String object, "This is a Java literal string".
* Because Java class instances are implicitly deallocated (there is no explicit deallocation operator), there cannot be dangling references in Java.
* All variables in the object-oriented languages Smalltalk, Python, Ruby, and Lua are references. They are always implicitly dereferenced.
* **6.9.7. Evaluation:**
* Pointers have been compared with the goto. The goto statement widens the range of statements that can be executed next. Pointer variables widen the range of memory cells that can be referenced by a variable.
* On the other hand, pointers are essential in some kinds of programming applications. For example, pointers are necessary to write device drivers, in which specific absolute addresses must be accessed.
* **6.9.8. Implementation of Pointers and Reference Types:**
* In most languages, pointers are used in heap management. The same is true for Java and C# references, as well as the variables in Smalltalk and Ruby, so we cannot treat pointers and references separately.
* **6.9.8.1. Representation of Pointers and References:**
* In most larger computers, pointers and references are single values stored in memory cells. However, in early microcomputers based on Intel microprocessors,
* addresses have two parts: a segment and an offset.
* **6.9.8.2. Solutions to the Dangling –Pointer Problem:**
* There have been several proposed solutions to the dangling-pointer problem. Among these are **tombstones** (Lomet, 1975), in which every heap-dynamic
* variable includes a special cell, called a tombstone, that is itself a pointer to the heap-dynamic variable. The actual pointer variable points only at tombstones and never to heap-dynamic variables. When a heap-dynamic variable is deallocated, the tombstone remains but is set to nil, indicating that the heap-dynamic
* variable no longer exists. This approach prevents a pointer from ever pointing to a deallocated variable. Any reference to any pointer that points to a nil tombstone can be detected as an error. Tombstones are costly in both time and space. Because tombstones are never deallocated, their storage is never reclaimed.
* An alternative to tombstones is the **locks-and-keys approach** used in the implementation of UW-Pascal (Fischer and LeBlanc, 1977, 1980). In this compiler, pointer values are represented as ordered pairs (key, address), where the key is an integer value. Heap-dynamic variables are represented as the storage for the variable plus a header cell that stores an integer lock value. When a heap-dynamic variable is allocated, a lock value is created and placed both in the lock cell of the heap-dynamic variable and in the key cell of the pointer that is specified in the call to **new**. Every access to the dereferenced pointer
* compares the key value of the pointer to the lock value in the heap-dynamic variable. If they match, the access is legal; otherwise the access is treated as a run-time error. Any copies of the pointer value to other pointers must copy the key value. Therefore, any number of pointers can reference a given heapdynamic
* variable. When a heap-dynamic variable is deallocated with **dispose**, its lock value is cleared to an illegal lock value. Then, if a pointer other than the one specified in the **dispose** is dereferenced, its address value will still be intact, but its key value will no longer match the lock, so the access will not be allowed.
* The best solution to the dangling-pointer problem is to take deallocation of heap-dynamic variables out of the hands of programmers. If programs cannot explicitly deallocate heap-dynamic variables, there will be no dangling pointers.
* **6.9.8.3. Heap Management:**
* Heap management can be a very complex run-time process. We examine the
* process in two separate situations: one in which all heap storage is allocated and deallocated in units of a single size, and one in which variable-size segments are allocated and deallocated.
* ***Single-Size Cells:*** The simplest situation is when all allocation and deallocation is of single-size cells. It is further simplified when every cell already contains a pointer.
* In a single-size allocation heap, all available cells are linked together using the pointers in the cells, forming a list of available space. Allocation is a simple matter of taking the required number of cells from this list when they are needed. Deallocation is a much more complex process. A heap-dynamic
* variable can be pointed to by more than one pointer, making it difficult to determine when the variable is no longer useful to the program. Simply because one pointer is disconnected from a cell obviously does not make it garbage; there could be several other pointers still pointing to the cell.
* There are several different approaches to garbage collection. The two most common traditional techniques are in some ways opposite processes. These are named **reference counters**, in which reclamation is incremental and is done when inaccessible cells are created, and **mark-sweep**, in which reclamation occurs only when the list of available space becomes empty. These two methods are sometimes called the **eager approach** and the **lazy approach**, respectively.
* ***Variable-Size Cells:*** Managing a heap from which variable-size cells are allocated has all the difficulties of managing one for single-size cells, but also has additional problems. Unfortunately, variable-size cells are required by most programming languages.

**UNIT -2**

**CHAPTER -2**

**EXPRESSIONS AND ASSIGNMENT STATEMENTS**

* 7.1 Introduction
* 7.2 Arithmetic Expressions
* 7.3 Type Conversions
* 7.4 Relational and Boolean Expressions
* 7.5 Short-Circuit Evaluation
* 7.6 Assignment Statements
* 7.7 Mixed-Mode Assignment
* **7.1 Introduction:**
* Expressions are the fundamental means of specifying computations in a programming language.
* It is crucial for a programmer to understand both the syntax and semantics of expressions of the language being used.
* To understand expression evaluation, it is necessary to be familiar with the orders of operator and operand evaluation.
* The operator evaluation order of expressions is dictated by the associativity and precedence rules of the language.
* Although the value of an expression sometimes depends on the order of operand and evaluation in expressions
* **7.2** **Arithmetic Expressions:**
* In programming languages, arithmetic expressions consist of operators, operands, parentheses, and function calls.
* An operator can be **unary**, meaning it has a single operand,
* **binary**, meaning it has two operands, or
* **ternary**, meaning it has three operands.
* In most programming languages, binary operators are **infix**, which means they appear between their operands.
* An implementation of such a computation must cause two actions:
* 1.Fetching the operands, usually from memory, and
* 2.Executing arithmetic operations on those operands.

Design Issues

* • What are the operator precedence rules?
* • What are the operator associativity rules?
* • What 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?
* **7.2.1 Operator Evaluation Order**
* The operator precedence and associativity rules of a language dictate the order of evaluation of its operators.
* **7.2.1.1 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 the multiplication), the result is 35.
* If evaluated right to left, the result is 23.
* For example, in mathematics, multiplication is considered to be of higher priority than addition, perhaps due to its 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 rules for expressions are based on the hierarchy of operator priorities, as designed by the language designer.
* The operator precedence rules of the common imperative languages are nearly all the same, because they are based on those of mathematics.
* In these languages exponentiation has the highest precedence followed by multiplication and division on the same level, followed by binary addition and subtraction on the same level.
* Many languages also include unary versions of addition and subtraction.
* Unary addition is called the **identity operator** because it usually has no associated operation and thus has no effect on its operand.
* In all of the common imperative languages, the unary minus operator can appear in an expression either at the beginning or anywhere inside the expression.
* 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
* In the first two cases, the relative precedence of the unary minus operator and the binary operator is irrelevant—the order of evaluation of the two operators has no effect on the value of the expression.
* In the last case, however, it does matter.Of the common programming languages, only Fortran, Ruby, Visual Basic, and Ada have the exponentiation operator.
* In all four, exponentiation has higher precedence than unary minus, so
  + -A \*\* B
* is equivalent to
  + -(A \*\* B)
* The precedences of the arithmetic operators of Ruby and the C-based 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 languages and ruby is exactly like the **rem** operator of Ada.
* The **Ada** mod operator is identical to **rem** when both operands are positive, but can be different when one or both are negative.
* **7.2.1.2 Associativity**
* Consider the following expression:
  + a - b + c - d
* If the addition and subtraction operators have the same level of precedence, the precedence rules say nothing about the order of evaluation of the operators.
* When an expression contains two adjacent occurrences of operators with the same level of precedence, **associativity** rules of the language will answer such questions.
* An operator can have either left or right associativity,
* Meaning that when there are two adjacent operators with the same precedence, the left operator is evaluated first or the right operator is evaluated first, respectively.
* Associativity in common languages is left to right, except that the exponentiation operator sometimes associates right to left. In the Java expression
  + a - b + c
* the left operator is evaluated first.
* Exponentiation in Fortran and Ruby is right associative, so in the expression
  + A \*\* B \*\* C
* the right operator is evaluated first.
* In Ada, exponentiation is nonassociative, which means that the expression
  + A \*\* B \*\* C
* is illegal. Such an expression must be parenthesized to show the desired order as in either
  + (A \*\* B) \*\* C
* or
  + A \*\* (B \*\* C)
* In Visual Basic, the exponentiation operator, ^, is left associative. The associativity rules for a few common languages are given here:



* In APL, all operators have the same level of precedence. Thus, the order of evaluation of operators in APL expressions is determined entirely by the associativity rule, which is right to left for all operators.
* For example, in the expression
  + A × B + C
* The addition operator is evaluated first, followed by the multiplication operator

(\* is the APL multiplication operator).

* If A were 3, B were 4, and C were 5, then the value of this APL expression would be 27.
* For example, addition is mathematically associative, so in mathematics the value of the expression
  + A + B + C
* does not depend on the order of operator evaluation.
* For example, there are pathological situations in which integer addition on a computer is *not* associative.
* For example, suppose that a program must evaluate the expression
  + A + B + C + D
* Where A and C are very large positive numbers, and B and D are negative numbers with very large absolute values.
* In this situation, adding B to A does not cause an overflow exception, but adding C to A does.
* Likewise, adding C to B does not cause overflow, but adding D to B does.
* Because of the limitations of computer arithmetic, addition is catastrophically nonassociative in this case.
* Therefore, if the compiler reorders these addition operations, it affects the value of the expression.
* This problem, of course, can be avoided by the programmer, assuming the approximate values of the variables are known.
* The programmer can specify the expression in two parts (in two assignment statements), ensuring that overflow is avoided.
* **7.2.1.3 Parentheses:**
* Programmers can alter the precedence and associativity rules by placing parentheses in expressions.
* A parenthesized part of an expression has precedence over its adjacent unparenthesized parts.
* For example, although multiplication has precedence over addition, in the expression
  + (A + B) \* C
* The addition will be evaluated first. Mathematically, this is perfectly natural.
* In this expression, the first operand of the multiplication operator is not available until the addition in the parenthesized subexpression is evaluated.
* Also, the expression from Section 7.2.1.2 could be specified as
  + (A + B) + (C + D)
* to avoid overflow.
* Languages that allow parentheses in arithmetic expressions could dispense with all precedence rules and simply associate all operators left to right or right to left.
* The programmer would specify the desired order of evaluation with parentheses.
* The advantage of this approach would be simple because neither the author nor the readers of programs would need to remember any precedence or associativity rules.
* The disadvantage of this scheme is that it makes writing expressions more tedious, and it also seriously compromises the readability of the code.
* **7.2.1.4 Ruby Expressions:**
* Ruby is a pure object-oriented language, which means, among other things, that every data value, including literals, is an object.
* Ruby supports the collection of arithmetic and logic operations that are included in the C-based languages.
* What sets Ruby apart from the C-based languages in the area of expressions is that all of the arithmetic, relational, and assignment operators, as well as array indexing, shifts, and bitwise logic operators, are implemented as methods.
* For example, the expression **a + b** is a call to the **+** method of the object referenced by **a**, passing the object referenced by **b** as a parameter.
* One interesting result of the implementation of operators as methods is that they can be overridden by application programs.
* **7.2.1.5 Conditional Expressions:**
* **if-then-else** statements can be 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**
* where expression\_1 is interpreted as a Boolean expression.
* If expression\_1 evaluates to true, the value of the whole expression is the value of expression\_2; otherwise, it is the value of expression\_3.
* For example, the effect of the example **if-then-else** can be achieved with the following assignment statement, using a conditional expression:
  + **average = (count == 0) ? 0 : sum / count;**
* In effect, the question mark denotes the beginning of the **then** clause, and the colon marks the beginning of the **else** clause. Both clauses are mandatory.
* Note that **?** is used in conditional expressions as a ternary operator.
* Conditional expressions can be used anywhere in a program (in a C-based language) where any other expression can be used.
* In addition to the C-based languages, conditional expressions are provided in Perl, JavaScript, and Ruby.
* **7.2.2 Operand Evaluation Order:**
* Variables in expressions are evaluated by fetching their values from memory.
* Constants are sometimes evaluated the same way. In other cases, a constant may be part of the machine language instruction and not require a memory fetch.
* If an operand is a parenthesized expression, all of the operators it contains must be evaluated before its value can be used as an operand.
* If neither of the operands of an operator has side effects, then operand evaluation order is irrelevant.
* Therefore, the only interesting case arises when the evaluation of an operand does have side effects.
* **7.2.2.1 Side Effects:**
* A **side effect** of a function, naturally called a functional side effect, occurs when

the function changes either one of its parameters or a global variable.

(A globalvariable is declared outside the function but is accessible in the function.)

* Consider the expression
  + a + fun(a)
* If fun does not have the side effect of changing a, then the order of evaluation of the two operands, a and fun(a), has no effect on the value of the expression.
* However, if fun changes a, there is an effect.
* Consider the following situation: fun returns 10 and changes the value of its parameter to 20. Suppose we have the following:
  + a = 10;
  + b = a + fun(a);
* Then, if the value of a is fetched first (in the expression evaluation process), its value is 10 and the value of the expression is 20.
* But if the second operand is evaluated first, then the value of the first operand is 20 and the value of the expression is 30.
* The following C program illustrates the same problem when a function changes a global variable that appears in an expression:
* **int** a = 5;
* **int** fun1() {
* a = 17;
* **return** 3;
* } /\* end of fun1 \*/
* **void** main() {
* a = a + fun1();
* } /\* end of main \*/
* The value computed for a in main depends on the order of evaluation of the operands in the expression a + fun1().
* The value of a will be either 8 (if a is evaluated first) or 20 (if the function call is evaluated first).
* Note that functions in mathematics do not have side effects, because there is no notion of variables in mathematics.
* There are 2 possible solutions to the problem of operand evaluation order and side effects.
* First, the language designer could disallow function evaluation from affecting the value of expressions by simply disallowing functional side effects.
* Second, the language definition could state that operands in expressions are to be evaluated in a particular order.
* Disallowing functional side effects in the imperative languages is difficult, and it eliminates some flexibility for the programmer.
* Access to globals in functions would also have to be disallowed.
* However, when efficiency is important, using access to global variables to avoid parameter passing is an important method of increasing execution speed.
* Therefore, no perfect solution, as is borne out by actual language designs.
* The Java language definition guarantees that operands appear to be evaluated in left-to-right order, eliminating the problem discussed in this section.
* **7.2.2.2 Referential Transparency and Side Effects:**
* The concept of referential transparency is related to and affected by functional side effects.
* A program has the property of **referential transparency** if any two expressions in the program that have the same value can be substituted for one another anywhere in the program, without affecting the action of the program.
* The value of a referentially transparent function depends entirely on its parameters.
* The connection of referential transparency and functional side effects is illustrated by the following example:
* result1 = (fun(a) + b) / (fun(a) - c);
* temp = fun(a);
* result2 = (temp + b) / (temp - c);
* If the function fun has no side effects, result1 and result2 will be equal, because the expressions assigned to them are equivalent.
* However, suppose fun has the side effect of adding 1 to either b or c.
* Then result1 would not be equal to result2. So, that side effect violates the referential transparency of the program in which the code appears.
* There are advantages to referentially transparent programs.
* The most important of these is that the semantics of such programs is much easier

to understand than the semantics of programs that are not referentially transparent.

* **7.3 Type Conversions**
* Type conversions are either narrowing or widening.
* A **narrowing conversion** converts a value to a type that cannot store even approximations of all of the values of the original type.
* For example, converting a double to a float in Java is a narrowing conversion, because the range of double is much larger than that of float.
* A **widening conversion** converts a value to a type that can include at least approximations of all of the values of the original type.
* For example, converting an int to a float in Java is a widening conversion.
* Widening conversions are nearly always safe, meaning that the magnitude of the converted value is maintained.
* Narrowing conversions are not always safe— sometimes the magnitude of the converted value is changed in the process.
* 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 issue of widening and narrowing conversions is relatively simple for the primitive numeric types.
* In java the following are widening conversions for primitive numeric types.
* byte TO short, int, long, float or double
* short TO int, long, float or double
* char TO int, long, float or double
* int TO long, float or double
* long TO float or double
* float TO double
* In java the following are narrowing conversions for primitive numeric types.
* 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

* Although widening conversions are usually safe, they can result in reduced accuracy.
* In many language implementations, although integer-to-floating-point conversions are widening conversions, some precision may be lost.
* For example, in many cases, integers are stored in 32 bits, which allows at least 9 decimal digits of precision.
* But floating-point values are also stored in 32 bits, with only about 7 decimal digits of precision (because of the space used for the exponent).
* So, integer-to-floating-point widening can result in the loss of two digits of precision.
* Type conversions can be either explicit or implicit.
* **7.3.1 Coercion in Expressions**
* One of the design decisions concerning arithmetic expressions is whether an operator can have operands of different types.
* Languages that allow such expressions, which are called **mixed-mode expressions**.
* Coercion was defined as an implicit type conversion that is initiated by the compiler.
* Type conversions explicitly requested by the programmer are referred to as explicit conversions, or casts, not coercions.
* When the 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.
* As a simple illustration of the problem, consider the following Java code:
* **int** a;
* **float** b, c, d;
* . . .
* d = b \* a;
* Assume that the second operand of the multiplication operator was supposed to be c, but because of a keying error it was typed as a.
* Because mixed-mode expressions are legal in Java, the compiler would not detect this as an error.
* It would simply insert code to coerce the value of the **int** operand, a, to **float**.
* If mixed-mode expressions were not legal in Java, this keying error would have been detected by the compiler as a type error.
* Because error detection is reduced when mixed-mode expressions are allowed.
* Ada allows very few mixed type operands in expressions.
* It does not allow mixing of integer and floating-point operands in an expression, with one exception:
* The exponentiation operator, \*\*, can take either a floating-point or an integer type for the first operand and an integer type for the second operand.
* If the Java code example were written in Ada, as in
* A : Integer;
* B, C, D : Float;
* . . .
* C := B \* A;
* Then the Ada compiler would find the expression erroneous, because Float and Integer operands cannot be mixed for the \* operator.
* ML and F# do not coerce operands in expressions. Any necessary conversions must be explicit.
* **7.3.2 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 conversion results in a significant change to the value of the object being converted.
* **casts**. To specify a cast, the desired type is placed in parentheses just before the expression to be converted, as in
* **(int)** angle
* One of the reasons for the parentheses around the type name in these conversions is that the first of these languages, C, has several two-word type names, such as **long int**.
* In ML and F#, the casts have the syntax of function calls.
* For example, in F# we could have the following:
* **float**(sum)
* **7.3.3 Errors in Expressions:**
* A number of errors can occur during expression evaluation.
* If the language requires type checking, either static or dynamic, then operand type errors cannot occur.
* 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.
* 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.
* One limitation of arithmetic is that division by zero is disallowed.
* Floating-point overflow, underflow, and division by zero are examples of run-time errors, which are sometimes called **exceptions**.
* **7.4 Relational and Boolean Expressions:**
* In addition to arithmetic expressions, programming languages support relational and Boolean expressions.
* **7.4.1 Relational Expressions:**
* A **relational operator** is an operator that compares the values of its two operands.
* A relational expression has two operands and one relational operator.
* The value of a relational expression is Boolean,
* The relational operators are often overloaded for a variety of types.
* The operation that determines the truth or falsehood of a relational expression depends on the operand types.
* It can be simple, as for integer operands, or complex, as for character string operands.
* Typically, the types of the operands that can be used for relational operators are numeric types, strings, and ordinal types.
* The syntax of the relational operators for equality and inequality differs among some programming languages.



* JavaScript and PHP have two additional relational operators, === and !==.
* These are similar to their relatives, == and !=, but prevent their operands from being coerced.
* For example, the expression "7" == 7 is true in JavaScript, because when a string and a number are the operands of a relational operator, the string is coerced to a number.
* However, "7" === 7 is false, because no coercion is done on the operands of this operator.
* Ruby uses == for the equality relational operator that uses coercions, and eql?
* Ruby uses === only in the **when** clause of its **case** statement.
* The relational operators always have lower precedence than the arithmetic operators, so that in expressions such as
* a + 1 > 2 \* b
* The arithmetic expressions are evaluated first.
* **7.4.2 Boolean Expressions:**
* Boolean expressions consist of Boolean variables, Boolean constants, relational expressions, and Boolean operators.
* The operators usually include those for the AND, OR, and NOT operations, and sometimes for exclusive OR and equivalence.
* Boolean operators usually take only Boolean operands (Boolean variables, Boolean literals, or relational expressions) and produce Boolean values.
* In the mathematics of Boolean algebras, the OR and AND operators must have equal precedence.
* In accordance with this, Ada’s AND and OR operators have equal precedence.
* However, the C-based languages assign a higher precedence to AND than OR.
* Perhaps this resulted from the baseless correlation of multiplication with AND and of addition with OR, which would naturally assign higher precedence to AND.
* Because arithmetic expressions can be the operands of relational expressions, and relational expressions can be the operands of Boolean expressions,
* The precedence of the arithmetic, relational, and Boolean operators in the C-based languages is as follows:

**

* Versions of C prior to C99 are odd among the popular imperative languages in that they have no Boolean type and thus no Boolean values.
* Instead, numeric values are used to represent Boolean values.
* In place of Boolean operands, scalar variables (numeric or character) and constants are used, with zero considered false and all nonzero values considered true.
* The result of evaluating such an expression is an integer, with the value 0 if false and 1 if true.
* Arithmetic expressions can also be used for Boolean expressions in C99 and C++.
* One odd result of C’s design of relational expressions is that the following expression is legal:
* a > b > c
* The leftmost relational operator is evaluated first because the relational operators of C are left associative, producing either 0 or 1.
* Then, this result is compared with the variable c. There is never a comparison between b and c in this expression.
* Some languages, including Perl and Ruby, provide two sets of the binary logic operators, **&&** and **and** for **AND** and **||** and **or** for **OR**.
* One difference between **&&** and **and** (and **||** and **or**) is that the spelled versions have lower precedence.
* Also, **and** and **or** have equal precedence, but && has higher precedence than **||**.
* **7.5 Short-Circuit Evaluation:**
* A **short-circuit evaluation** of an expression is one in which the result is determined without evaluating all of the operands and/or operators.
* For example,the value of the arithmetic expression
* (13 \* a) \* (b / 13 - 1)
* is independent of the value of (b / 13 - 1) if a is 0, because 0 \* x = 0 for any x.
* So, when a is 0, there is no need to evaluate (b / 13 - 1) or perform the second multiplication.
* However, in arithmetic expressions, this shortcut is not easily detected during execution, so it is never taken.
* The value of the Boolean expression
* (a >= 0) && (b < 10)
* is independent of the second relational expression if a < 0, because the expression (FALSE && (b < 10)) is FALSE for all values of b.
* So, when a < 0, there is no need to evaluate b, the constant 10, the second relational expression, or the && operation.
* Unlike the case of arithmetic expressions this shortcut can be easily discovered during execution.
* Java did not use short-circuit evaluation.
* A table lookup loop could be written using the **while** statement. One simple version of Java code for such a lookup, assuming that list, which has listlen elements, is the array to be searched and key is the searched-for value, is
* index = 0;
* **while** ((index < listlen) && (list[index] != key))
* index = index + 1;
* If evaluation is not short-circuit, both relational expressions in the Boolean expression of the **while** statement are evaluated, regardless of the value of the first.
* Thus, if key is not in list, the program will terminate with a subscript out-of-range exception.
* The same iteration that has index == listlen will reference list[listlen], which causes the indexing error because list is declared to have listlen-1 as an upper-bound subscript value.
* If a language provides short-circuit evaluation of Boolean expressions and it is used, this is not a problem.
* In the preceding example, a short-circuit evaluation scheme would evaluate the first operand of the AND operator, but it would skip the second operand if the first operand is false.
* A language that provides short-circuit evaluations of Boolean expressions and also has side effects in expressions allows subtle errors to occur.
* **7.6 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.
* **7.6.1 Simple Assignments:**
* The general syntax of the simple assignment statement is
* **<target\_variable> <assignment\_operator> <expression>**
* Nearly all programming languages currently being used use 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 confusion 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 design choices of how assignments are used in a language have varied widely.
* In some languages, such as Fortran and Ada, an assignment can appear only as a stand-alone statement, and the destination is restricted to a single variable.
* **7.6.2 Conditional Targets:**
* C++ allows conditional targets on assignment statements. For example, consider
* flag ? count1 : count2 = 0;
* which is equivalent to

**if** (flag)

{

count1 = 0;

}

**else**

{

count2 = 0;

}

* **7.6.3 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, as in
* a = a + b
* Compound assignment operators were introduced by ALGOL 68, were later adopted in a slightly different form by C, and are part of the other C-based languages, as well as Perl, JavaScript, Python, and Ruby.
* 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;
* The languages that support compound assignment operators have versions for most of their binary operators.
* **7.6.4 Unary Assignment Operators**
* The C-based languages, Perl, and JavaScript include two special unary arithmetic operators that are actually abbreviated assignments.
* They combine increment and decrement operations with assignment.
* The operators ++ for increment, and – – for decrement, can be used either in expressions or to form stand-alone single-operator assignment statements.
* They can appear either as prefix operators, meaning that they precede the operands, or as

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 also 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;
* An example of the use of the unary increment operator to form a complete assignment statement is
* count ++;
* which simply increments count.
* It does not look like an assignment, but it certainly is one. It is equivalent to the statement
* count = count + 1;
* When two unary operators apply to the same operand, the association is right to left.
* For example, in
* -count ++
* count is first incremented and then negated. So, it is equivalent to
* -(count ++)
* rather than
* (- count) ++
* **7.6.5 Assignment as an Expression**
* In the C-based 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.
* The result, or value assigned, is then compared with the constant EOF.
* If ch is not equal to EOF, the compound statement {...} is executed.
* Note that the assignment must be parenthesized—in the languages that support assignment as an expression,
* The precedence of the assignment operator is lower than that of the relational operators.
* Without the parentheses, the new character would be compared with EOF first.
* Then, the result of that comparison, either 0 or 1, would be assigned to ch.
* The disadvantage of allowing assignment statements to be operands in expressions is that it provides yet another kind of expression side effect.
* This type of side effect can lead to expressions that are difficult to read and understand.
* For example, the expression
* a = b + (c = d / b) - 1
* denotes the instructions
* Assign d / b to c
* Assign b + c to temp
* Assign temp - 1 to a
* Note that the treatment of the assignment operator as any other binary operator allows the effect of multiple-target assignments, such as
* sum = count = 0;
* In which count is first assigned the zero, and then count’s value is assigned to sum.
* This form of multiple-target assignments is also legal in Python
* In particular, if we type

**if** (x = y) ...

* instead of

**if** (x == y) ...

* which is an easily made mistake, it is not detectable as an error by the compiler.
* Rather than testing a relational expression, the value that is assigned to x is tested.
* **7.6.6 List Assignments:**
* Several recent programming languages, including Perl, Ruby, and 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 $third.
* If the values of two variables must be interchanged, this can be done with a single assignment, as with
* ($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).
* The syntax of the simplest form of Ruby’s multiple assignment is similar to that of Perl, except the left and right sides are not parenthesized.
* **7.7 Mixed-Mode Assignment:**
* An operator can have operands of different types.
* Assignment statements also are mixed mode. The design question is:
* 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 type mismatch?

* Fortran, C, C++, and Perl use coercion rules for mixed-mode assignment that are similar to those they use for mixed-mode expressions;
* 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.
* So, an **int** value can be assigned to a **float** variable, but not vice versa.
* Disallowing half of the possible mixed-mode assignments is a simple but effective way to increase the reliability of Java and C#, relative to C and C++.
* In all languages that allow mixed-mode assignment , the coercion takes place only after the right side expression has been evaluated.
* For example consider the following code
* int a, b
* float c;
* ..
* c = a / b;
* Because c is float , the value of a and b could be coerced to float before the division, which could produce a different values for c

**UNIT -2**

**CHAPTER -3**

**STATEMENT-LEVEL CONTROL STRUCTURES**

* 8.1 Introduction
* 8.2 Selection Statements
* 8.3 Iterative Statements
* 8.4 Unconditional Branching
* 8.5 Guarded Commands
* **8.1 Introduction:**
* **Control structures**
* Computations in imperative-language programs are accomplished by evaluating expressions and assigning the resulting values to variables. However, there are few useful programs that consist entirely of assignment statements. At least two additional linguistic mechanisms are necessary to make the computations in programs flexible and powerful: some means of selecting among alternative control flow paths (of statement execution) and some means of causing the repeated execution of statements or sequences of statements. Statements that provide these kinds of capabilities are called **control statements**. The control statements of the first successful programming language, Fortran.
* **8.2 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: two-way and n-way, or multiple selection.
* **8.2.1. Two-Way Selection Statements:**
* The general form of a two-way selector is as follows:
* **if** control\_expression
* then clause
* else clause
* **8.2.1.1. Design Issues:** The design issues for two-way selectors can be summarized 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?
* **8.2.1.2. The Control Expression:**
* Control expressions are specified in parentheses if the **then** reserved word (or some other syntactic marker) is not used to introduce the then clause. In those cases where the **then** reserved word (or alternative marker) is used, there is less need for the parentheses, so they are often omitted, as in Ruby.
* **8.2.1.3. 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. Many languages use braces to form compound statements, which serve as the bodies of then and else clauses. In Fortran 95, Ada, Python, and Ruby, the then and else clauses are statement sequences, rather than compound statements. The complete selection statement is terminated in these languages with a reserved word.
* Python uses indentation to specify compound statements. For example,
* **if** x > y :
* x = y
* print "case 1"
* All statements equally indented are included in the compound statement.2
* Notice that rather than **then**, a colon is used to introduce the then clause in
* Python.

* **8.2.1.4. Nesting Selectors:** we discussed the problem of syntactic ambiguity of a straightforward grammar for a two-way selector statement. That ambiguous grammar was as follows:
* 
* The issue was that 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. This problem is reflected in the semantics of selection statements. Consider the following Java-like code:
* **if** (sum == 0)
* **if** (count == 0)
* result = 0;
* **else**
* result = 1;
* This statement 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.
* The crux of the problem in this example is that the else clause follows two then clauses with no intervening else clause, and there is no syntactic indicator to specify a matching of the else clause to one of the then clauses. In Java, as in many other imperative languages, the static semantics of the language specify that the else clause is always paired with the nearest previous unpaired then clause.
* The disadvantage of using a rule rather than some syntactic entity is that although the programmer may have meant the else clause to be the alternative to the first then clause and the compiler found the structure syntactically correct, its semantics is the opposite. To force the alternative semantics
* in Java, the inner **if** is put in a compound, as in
* **if** (sum == 0) {
* **if** (count == 0)
* result = 0;
* }
* **else**
* result = 1;
* C, C++, and C# have the same problem as Java with selection statement nesting. Because Perl requires that all then and else clauses be compound, it does not. In Perl, the previous code would be written as
* **if** (sum == 0) {
* **if** (count == 0) {
* result = 0;
* }
* } **else** {
* result = 1;
* }
* If the alternative semantics were needed, it would be
* **if** (sum == 0) {
* **if** (count == 0) {
* result = 0;
* }
* **else** {
* result = 1;
* }
* }
* In Ruby, the then and else clauses consist of statement sequences rather than compound statements. The first interpretation of the selector example at the beginning of this section, in which the else clause is matched to the nested **if**, can be written in Ruby as follows:
* **if** sum == 0 **then**
* **if** count == 0 **then**
* result = 0
* **else**
* result = 1
* **end**
* **end**
* Because the **end** reserved word closes the nested **if**, it is clear that the else clause is matched to the inner then clause. The second interpretation of the selection statement at the beginning of this section, in which the else clause is matched to the outer **if**, can be written in Ruby as follows:
* **if** sum == 0 **then**
* **if** count == 0 **then**
* result = 0
* **end**
* **else**
* result = 1
* **end**
* The following statement, written in Python, is semantically equivalent to the last Ruby statement above:
* **if** sum == 0 :
* **if** count == 0 :
* result = 0
* **else**:
* result = 1
* If the line **else:** were indented to begin in the same column as the nested **if**, the else clause would be matched with the inner **if**.
* **8.2.2. Multiple-Selection Statements:**
* The **multiple-selection** statement allows the selection of one of any number of statements or statement groups. It is, therefore, a generalization of a selector.
* **8.2.1. Design Issues:** The following is a summary of these design issues:
* • What is the form and type of the expression that controls the selection?
* • How are the selectable segments specified?
* • Is execution flow through the structure restricted to include just a single selectable segment?
* • How are the case values specified?
* • How should unrepresented selector expression values be handled, if at all?
* **8.2.2. Examples of Multiple Selectors:**
* The C multiple-selector statement, **switch**, which is also part of C++, Java, and JavaScript, is a relatively primitive design. Its general form is
* **switch** (expression) {
* **case** constant\_expression1:statement1;
* . . .
* **case** constantn: statement\_n;
* [**default**: statementn+1]
* }
* where the control expression and the constant expressions are some discrete type. This includes integer types, as well as characters and enumeration types. The selectable statements can be statement sequences, compound statements, or blocks. The optional **default** segment is for unrepresented values of the control expression. If the value of the control expression is not represented and no default segment is present, then the statement does nothing.
* The **switch** statement 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 executed every time the code at the 1 or 3 constants is executed. To separate these segments logically, an explicit branch must be included. The **break** statement, which is actually a restricted goto, is normally used for exiting **switch** statements.
* The following **switch** statement uses **break** to restrict each execution to a single selectable segment:
* **switch** (index) {
* **case** 1:
* **case** 3: odd += 1;
* sumodd += index;
* **break**;
* **case** 2:
* **case** 4: even += 1;
* sumeven += index;
* **break**;
* **default**: printf("Error in switch, index = %d\n", index);
* }
* It is convenient to allow control to flow from one selectable code segment to another. For example, in the example above, the segments for the case values 1 and 2 are empty, allowing control to flow to the segments for 3 and 4, respectively. This is the reason why there are no implicit branches in
* the **switch** statement. The reliability problem with this design arises when the mistaken absence of a **break** statement in a segment allows control to flow to the next segment incorrectly.
* The C# switch statement differs from that of its C-based predecessors in two ways. First, C# has a static semantics rule that disallows the implicit execution of more than one segment. The rule is that every selectable segment must end with an explicit unconditional branch statement: either a **break**, which transfers control out of the **switch** statement, or a goto, which can transfer control to one of the selectable segments (or virtually anywhere else).
* For example,
* **switch** (value) {
* **case** -1:
* Negatives++;
* **break**;
* **case** 0:
* Zeros++;
* **goto case** 1;
* **case** 1:
* Positives++;
* **default**:
* Console.WriteLine("Error in switch \n");
* }
* Note that Console.WriteLine is the method for displaying strings in C#.
* Ruby has two forms of multiple-selection constructs, both of which are called *case expressions* and both of which yield the value of the last expression evaluated. The only version of Ruby’s case expressions that is described here is semantically similar to a list of nested if statements:
* **case**
* **when** Boolean\_expression **then** expression
* . . .
* **when** Boolean\_expression **then** expression
* [**else** expression]
* **end**
* The semantics of this case expression is that the Boolean expressions are evaluated one at a time, top to bottom. The value of the case expression is the value of the first then expression whose Boolean expression is true. The else represents true in this statement, and the else clause is optional. For
* example,
* leap = **case**
* **when** year % 400 == 0 **then true**
* **when** year % 100 == 0 **then false**
* **else** year % 4 == 0
* **end**
* This case expression evaluates to true if year is a leap year.
* **8.2.3. Multiple Selection Using if:**
* a **switch** or **case** statement is inadequate for multiple selection (Ruby’s **case** is an exception). 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. To alleviate the poor readability of deeply nested two-way selectors, some languages, such as Perl and Python, have been extended specifically for this use. The nested selector is then called an **else-if clause**. Consider the following Python selector statement (note that else-if is spelled **elif** in Python):
* **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
* The else-if version (the first) is the more readable of the two. Notice that this example is not easily simulated with a **switch** statement, because each selectable statement is chosen on the basis of a Boolean expression. Therefore, the else-if statement is not a redundant form of **switch**.
* The Python example if-then-else-if statement above can be written as the Ruby **case** statement:
* **case**
* **when** count < 10 **then** bag1 = **true**
* **when** count < 100 **then** bag2 = **true**
* **when** count < 1000 **then** bag3 = **true**
* **end**
* Else-if statements are based on the common mathematics statement, the conditional expression.
* **8.3 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 statement is often called a **loop**.
* The **body** of an iterative statement is the collection of statements whose execution is controlled by the iteration statement. We use the term **pretest** to mean that the test for loop completion occurs before the loop body is executed and **posttest** to mean that it occurs after the loop body is executed. The iteration
* statement and the associated loop body together form an **iteration statement**.
* **8.3.1. 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 sequential loop variable values, often called the **stepsize**. The initial, terminal, and stepsize specifications of a loop are called the **loop parameters**.
* **8.3.1.1. Design Issues:** The following is a summary of these design issues:
* • What are the type and scope of the loop variable?
* • 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?
* **8.3.1.2. The Do Statement in FORTRAN 95:**
* A Do statement in Fortran uses a simple iterator over integer values. The general form of FORTRAN Do Statement as
* Do label variable := initial, terminal[,stepwise]
* ………………
* end Do
* For example, consider the following statement:
* Do Count = 1, 9, 2
* In this statement, 1 is the initial value of Count, 9 is the last value, and the step size between values is 2. An internal function, the iterator, must be called for each iteration to compute the next value of Count (by adding 2 to the last value of Count, in this example) and test whether the iteration should continue.

* **8.3.1.3. 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 **reverse** reserved word, when present, indicates that the values of the discrete range are assigned to the loop variable in reverse order.
* 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. For example, in
* 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.
* **8.3.1.4. The for Statement of the C-Based Languages:**
* The general form of C’s **for** statement is
* **for** (expression\_1; expression\_2; expression\_3)
* loop body
* The loop body can be a single statement, a compound statement, or a null statement.
* Because assignment statements in C produce results and thus can be considered expressions, the expressions in a **for** statement are often assignment 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.
* **for** (count = 1; count <= 10; count++)
* . . .
* }
* All of the expressions of C’s **for** are optional. An absent second expression is considered true, so a **for** without one is potentially an infinite loop. If the first and/or third expressions are absent, no assumptions are made. For example, if the first expression is absent, it simply means that no initialization takes place.
* The **for** statement of C99 and C++ differs from that of earlier versions of C in two ways. First, in addition to an arithmetic expression, it can use a Boolean expression for loop control. Second, the first expression can include variable definitions. For example,
* **for** (**int** count = 0; count < len; count++) { . . . }
* The scope of a variable defined in the **for** statement is from its definition to the end of the loop body.
* The **for** statement of Java and C# is like that of C++, except that the loop control expression is restricted to **boolean**.
* **8.3.1.5. The for Statement of Python:**
* The general form of Python’s **for** 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
* produces
* 2
* 4
* 6
* For most simple counting loops in Python, the **range** function is used. **range** takes one, two, or three parameters. The following examples demonstrate the actions of **range**:
* **range**(5) returns [0, 1, 2, 3, 4]
* **range**(2, 7) returns [2, 3, 4, 5, 6]
* **range**(0, 8, 2) returns [0, 2, 4, 6]
* **8.3.2. Logically Controlled Loops:** In many cases, collections of statements must be repeatedly executed, but the repetition control is based on a Boolean expression rather than a counter. For these situations, a logically controlled loop is convenient.
* **8.3.2.1. Design Issues:** Because they are much simpler than counter-controlled loops, logically controlled
* loops have fewer design issues.
* • Should the control be pretest or posttest?
* • Should the logically controlled loop be a special form of a counting loop or a separate statement?
* **8.3.2.2. 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 statement forms are exemplified by the following C# code segments:
* sum = 0;
* indat = Int32.Parse(Console.ReadLine());
* **while** (indat >= 0) {
* sum += indat;
* indat = Int32.Parse(Console.ReadLine());
* }
* value = Int32.Parse(Console.ReadLine());
* **do** {
* value /= 10;
* digits ++;
* } **while** (value > 0);
* Note that all variables in these examples are integer type. The Read- Line method of the Console object gets a line of text from the keyboard. Int32.Parse finds the number in its string parameter, converts it to **int** type, and returns it.
* In the pretest version of a logical loop (**while**), the statement or statement segment is executed as long as the expression evaluates to true. In the posttest version (**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 at least once. In both cases, the statement can be compound.
* Java’s **while** and **do** statements are similar to those of C and C++, except the control expression must be **boolean** type, and because Java does not have a goto, the loop bodies cannot be entered anywhere but at their beginnings.
* **8.3.3. 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 body.
* As a result, some languages provide this capability. A syntactic mechanism for user-located loop control can be relatively simple, so its design is not difficult. Such loops have the structure of infinite loops but include user-located loop exits.
* The design issues for such a mechanism are the following:
* • 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).
* Following is an 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;
* }
* C, C++, and Python include an unlabeled control statement, **continue**, that transfers control to the control mechanism of the smallest enclosing loop. This is not an exit but rather a way to skip the rest of the loop statements on the current iteration without terminating the loop structure. For example, consider the following:
* **while** (sum < 1000) {
* getnext(value);
* **if** (value < 0) **continue**;
* sum += value;
* }
* A negative value causes the assignment statement to be skipped, and control is transferred instead to the conditional at the top of the loop. On the other hand, in
* **while** (sum < 1000) {
* getnext(value);
* **if** (value < 0) **break**;
* sum += value;
* } a negative value terminates the loop.
* **8.3.4. Iteration Based on Data Structures:**
* A general data-based iteration statement uses a user-defined data structure and a user-defined function (the iterator) to go through the structure’s elements.
* The iterator is called at the beginning of each iteration, and each time it is called, the iterator returns an element from a particular data structure in some specific order.
* For example, suppose a program has a user-defined binary tree of data nodes, and the data in each node must be processed in some particular order.
* A user-defined iteration statement for the tree would successively set the loop variable to point to the nodes in the tree, one for each iteration.
* The initial execution of the user-defined iteration statement needs to issue a special call to the iterator to get the first tree element.
* The iterator must always remember which node it presented last so that it visits all nodes without visiting any node more than once. So an iterator must be history sensitive. A user-defined iteration statement terminates when the iterator fails to find more elements.
* The **for** statement of the C-based languages, because of its great flexibility, can be used to simulate a user-defined iteration statement. Once again, suppose the nodes of a binary tree are to be processed. If the tree root is pointed to by a variable named root, and if traverse is a function that sets its parameter to point to the next element of a tree in the desired order, the following could be used:
* **for** (ptr = root; ptr == null; ptr = traverse(ptr)) {
* . . .
* } In this statement, traverse is the iterator.
* **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 a program’s statements.
* A few languages have been designed without a goto—for example, Java, Python, and Ruby.
* The relatively new language, C#, includes a goto, even though one of the languages on which it is based, Java, does not.
* **Guarded Commands:-**
* New and quite different forms of selection and loop structures were suggested by Dijkstra (1975). His primary motivation was to provide control statements that would support a program design methodology that ensured correctness during development rather than when verifying or testing completed programs.
* Another motivation is the increased clarity in reasoning that is possible with guarded commands. Guarded commands are covered in this chapter because they are the basis for two linguistic mechanisms developed later for concurrent programming in two languages, CSP (Hoare, 1978) and Ada.
* Dijkstra’s selection statement has the form
* **if** <Boolean expression> -> <statement>
* [] <Boolean expression> -> <statement>
* [] . . .
* [] <Boolean expression> -> <statement>
* **fi**
* The closing reserved word, **fi**, is the opening reserved word spelled backward. This form of closing reserved word is taken from ALGOL 68. The small blocks, called *fatbars*, are used to separate the guarded clauses and allow the clauses to be statement sequences. Each line in the selection statement, consisting of a Boolean expression (a guard) and a statement or statement sequence, is called a **guarded command**.
* This selection statement has the appearance of a multiple selection, but its semantics is different. 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 statements can be nondeterministically chosen for execution. An implementation may always choose the statement associated with the first Boolean expression that evaluates to true. But it may choose any statement associated with a true Boolean expression. So, the correctness of the program cannot depend on which statement is chosen (among those associated with true Boolean expressions). If none of the Boolean expressions is true, a run-time error occurs that causes program termination.
* Consider the following example:
* **if** i = 0 -> sum := sum + i
* [] i > j -> sum := sum + j
* [] j > i -> sum := sum + i
* **fi**
* If i = 0 and j > i, this statement chooses nondeterministically between the first and third assignment statements. If i is equal to j and is not zero, a runtime error occurs because none of the conditions is true.
* The loop structure proposed by Dijkstra has the form
* **do** <Boolean expression> -> <statement>
* [] <Boolean expression> -> <statement>
* [] . . .
* [] <Boolean expression> -> <statement>
* **od**
* The semantics of this statement is that all Boolean expressions are evaluated on each iteration. If more than one is true, one of the associated statements is nondeterministically (perhaps randomly) chosen for execution, after which the expressions are again evaluated. When all expressions are simultaneously
* false, the loop terminates.
* Consider the following problem: Given four integer variables, q1, q2, q3, and q4, rearrange the values of the four so that q1 ≤ q2 ≤ q3 ≤ q4. Without guarded commands, one straightforward solution is to put the four values into an array, sort the array, and then assign the values from the array back into
* the scalar variables q1, q2, q3, and q4. While this solution is not difficult, it requires a good deal of code, especially if the sort process must be included. Now, consider the following code, which uses guarded commands to solve the same problem but in a more concise and elegant way.
* **do** q1 > q2 -> temp := q1; q1 := q2; q2 := temp;
* [] q2 > q3 -> temp := q2; q2 := q3; q3 := temp;
* [] q3 > q4 -> temp := q3; q3 := q4; q4 := temp;
* **od**