3.1 INTRODUCTION

Imperative languages are abstractions of Von-Neumann architecture. The architecture has two components they are,

(1) **Memory :** It stores both instructions and data.

(2) **Processor :** It provides operations for modifying the contents of memory.

A variable can be characterized by attributes. The most important of which is type, a fundamental concept in programming languages. The design of datatypes of a language it requires a different issues can be considered.

Among the most important issues are the scope and lifetime of variables. Related to these are the issues of type checking and initialization. Type equivalence is another important part of the datatype design of a language.

3.2 NAMES

Names, also known as identifiers, are usually related to the subprograms, labels formal parameters and other programming constructs.

3.2.1 Design Issues

Following are the design issues associated with names,

(1) Can the names be case-sensitive?

Names in most of the programming languages begin with a letter followed by a string of letters, digits and underscore character. C-based languages are case-sensitive i.e., the upper-case and lower-case letters are easily distinguishable.

(2) Does the special words belongs to the category of reserved words or keywords?

3.2.2 Name Forms

Programming languages use case-sensitive names hence, in languages like C, both the names written in upper-case and the names written in lower-case are totally distinct from each other.

Example : The three names given below are totally distinct,

shaz, Shaz, SHAZ.

In languages such as C# and Java, this problem of case-sensitivity, associated with names can't be avoided because the predefined words are themselves composed of both upper-case and lower-case letters. A string can be transformed into its equivalent integer number by using a method called parseInt.

Advantages

(1) Case-sensitive names help in enhancing the writability criterion.

(2) They maintain code readability and provides clarity.

(3) It expands the limit of using same name in different ways.

Disadvantages : The potential danger of using case-sensitive names are as follows,

(1) The main problem associated with case-sensitive names is of writability rather than that of readability. Because of the use of case-sensitive names, the programmers finds it difficult to write correct programs.

(2) Care must be taken while performing string processing operations by the programmers.

(3) The problems may arises when the right case format is not used for providing input data in the case-sensitive languages.

(4) Assigning names to the user-defined variables must be done carefully because the compiler will results-in an error if the user-defined variables coincides with the predefined variables.

3.2.3 Special Words

Special words in programing languages enhances the readability of the programs. In many of the programming languages, these words are categorized as reserved words while in other they are only keywords.

3.2.3.1 Keywords

A keyword is actually a programming language word that has a special meaning only under certain contexts. Fortran is an example of such a language.

Example : In Fortran, when the word real appears at the start of a statement followed by a name, it is regarded as a keyword which specifies the statement as a declarative statement. On the other hand, if the assignment operator follows real it is treated as a variable name.

These two applications are illustrated as follows,

```
Real pi

Real = 3.141
```

The differentiation between names and special words can be identified by Fortran compilers and programmers depending on their contexts.

Disadvantages

(1) Redefining keywords leads to readability problems hence, reserved words are better than the keywords.

(2) Syntactic analysis during the process of translation can be done in an easier way by using reserved words rather than the keywords.

3.2.3.1.1 Keywords Supported by C Language

Keywords refers to the special words whose meaning has already been defined to the C compiler.

C provides 32 keywords of which, some of them are,

(1) *'auto' Keyword :* An automatic variable can be declared using the auto storage-class specifier. An auto variable is visible only in the block in which it is declared. The variables of auto type are not initialized automatically but requires either an explicit initialization during their declarations or assigns initial values to them by using the statements within a block.

Example : auto int a;

(2) *'static' Keyword :* The 'static' keyword specifies that the variable has static duration i.e., it is allocated at the time of program start and is deallocated at the time of program termination. A static variable has an initial value '0' unless it is assigned with some value. When a variable is declared as static inside a function it retains its state between calls to that function.

Example : static int a;

(3) 'register' Keyword : It is used to define local variables that are to be stored in registers rather than in RAM. This indicates that the maximum size of a variable is equal to the size of a register in which it is stored. The unary '&' operator can't be applied to a 'register' variable. It is used for the variables that needs quick access.

Example : register int a;

(4) 'extern' Keyword : This keyword is used to define global variables that are visible in all the object modules i.e., a variable declared as 'extern' specifies that it has an external linkage. An 'extern' type variable during its modification specifies that the variable has static duration.

Example : extern int a;

(5) 'sizeof' Operator : It return the number of bytes occupied by a variable i.e., size of variables in bytes.

Example : sizeof(str);

(6) **'typedef' Keyword :** Every variable has a data type typedef is a keyword that is used to define new names for the data types thereby increasing the programs readability. It provides a precise and meaningful method of calling a data type.

Example

```
structure student
{
    char name[15];
    int rno;
    float avg;
};
typedef struct student st;
st s1, s2;
```

(7) **'break' Keyword :** This is used to take the control out of 'for', 'while', 'do-while' and 'switch' statements.

It must not be used to take the control out of 'if' block.

Example

```
switch(ch)
{
    case 'S' : printf("Sincere");
               break;
    case 'H' : printf("Honest");
               break;
    case 'A' : printf("Adorable");
               break;
    case 'Z' : printf("Zealous");
               break;
    default : printf("GOOD GIRL");
              break;
}
```

When a 'break' statement is not written then the successive execution of the cases takes place leading to an unexpected output.

(8) **'continue' Keyword :** This is actually a statement that takes control to the start of a loop by neglecting all the statements within the loop which have not yet been executed.

3.2.3.2 Reserved Word

A reserved word is a special word in any programming language which can't be used as an identifier. They're better than the keywords because redefinitions of the keywords causes readability problems.

Example : Consider the following statements in Fortran,

```
Integer Real
Real Integer
```

The above declarative statements are legal and declares the variable Real to be of Integer type and the variable Integer to be of Real type but it may lead to ambiguities.

In Ada, the built-in data types integer and float are already defined and are not reserved but they can be redefined by any Ada program.

3.3 VARIABLES

Variables are the abstractions in a language for the memory cells of a computer. Variables are not just names for the memory locations but refers to something else. The transformation from machine language to assembly languages leads to the replacement of numeric memory addresses with variable names, thus making programs more readable and easier to write and maintain.

A variable is associated with many attributes are,

(1) Name. (2) Address.

(3) Value. (4) Type.

3.3.1 Name

Variable names are the most common names that occur in a program. A name is actually a string of characters used to identify some program entity. Names in any programming language takes the form in which they consists of a letter at their start followed by a sequence of letters, digits and underscore characters.

Some programming languages imposes limit on the length of the variable names such as in Fortran 95, where only 31 characters are allowed for names whereas other languages such as Java and C++ have no length limit for names. In some languages like C, the variable names are case-sensitive which indicates that both the upper-case and lower-case letters are distinct from one another, hence the following three names in C are different, item, ITEM, Item. Languages such as C# is not a case-sensitive language.

Variable names are preceded by a keyword or a reserved word as shown in the following example,

```
int a;
float b;
```

In the above example, int and float are the data types which are reserved words and can't be used as variable names, 'a' and 'b' are the variable names.'

3.3.2 Address

The address of a variable refer to the memory address associated with it. In many programming languages one name can be associated with different addresses at different places in a program.

Example : Let fun1 and fun2 be the two methods in a Java program. Each of these functions defines a local variable with the same name, say 'a'. As the two variables are independent of each other, a reference to 'a' in 'fun1' is not related to 'a' in 'fun2'.

The address of a variable is called its 'l-value'. When more than one variable is accessing the same memory location, the variables are called Aliases. If two variables are Aliases to each other then any changes made to one causes the other to change. This affects the readability. Aliases can be created using data structures in C and C++.

3.3.2.1 Aliases

Two names are said to be Aliases when they point to the same memory location.

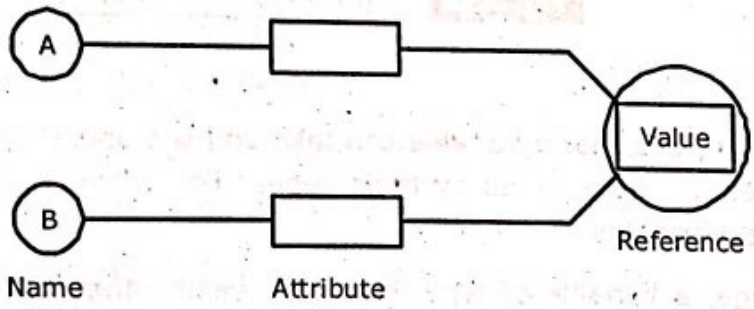


Fig. 3.3.1 Aliasing

In the Fig. 3.3.1, two variables A and B are pointing to the same memory location. Here A and B are called Aliases. Most of the programming languages allow aliasing by providing a relevant mechanism (For e.g., an Equivalence statement in Fortran).

Problems Associated with Aliasing

(1) Aliasing deters reliability because altering any one value causes the changes to be reflected in all aliases.

Example : If A and B are aliases (pointing to the same memory location) then any change made to A is reflected in B (in the sense B also changes).

(2) Aliasing permits only explicit changes in the program values but not the implicit changes.

(3) Program understanding and verification becomes difficult through Aliasing.

(4) Aliasing also make analysis and optimization of programs difficult.

(5) In C language, allowing pointers of different types to point (or refer) to the same memory location is illegal.

3.3.3 Value

Memory cells are the abstract cells used to store the values of the variables. The size of the physical memory cells is 1 byte which is not enough for many program variables. The size of an abstract cell depends on the size of the variable associated with it. For example, a value of type 'int' can occupy four bytes of memory in a specific language implementation and hence can be considered as an integer value occupying a single abstract memory cell. A variables value can be a single number, character or a pointer to some other objects.

Example : Consider an example in which a data object capable of storing 16 bits stores a value 10. This can be represented as shown in Fig. 3.3.2.

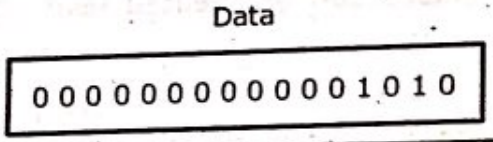


Fig. 3.3.2 Representation of Data

3.3.4 Type

The range of values that a variable can take and the operation that can be performed on these variables are determined by their 'types'. For example, int, char, float, byte are the types of the variables.

In C language, a variable of type 'int' has a value range of −32768 to +32767. The 'float' type in Java specifies a value range of -2^{31} to $2^{31} - 1$ and the operations such as addition, subtraction etc.

3.4 THE CONCEPT OF BINDING

An association between an attribute and entity is called binding. The time at which binding takes place is called binding time. The values of attributes must be set before they can be used. Binding can occur in various ways,

(1) **Language Definition (Design) Time Binding :** When a language is designed, certain issues have to be considered.

Example : The * operator is bound to multiplication at language design time.

(2) **Language Implementation Time Binding :** In most of the languages (including Fortran, Ada, C and C++) a set of values are bound to an integer type at language implementation time.

(3) **Compile-time (Translation Time) Binding :** A variable type is bound at compile time in Fortran, Pascal and C.

(4) **Linking Time Binding :** A subprogram call is bound to a subprogram code at link time.

(5) **Execution-time (or Runtime) Binding :** In most of the programming languages variables are bound to a value at execution time, and the binding can be modified repeatedly during execution.

The first four categories refer to binding before run time. These categories refer to static binding. Last category refers to dynamic binding.

For example, assuming a simple C code segment,

```
int C;
...
C = C - 8;
```

Some of the bindings and their binding times are,

(1) **Type of Count :** Bound at compile time.

(2) **Set of Possible Values for Count :** Bound at compiler design time.

(3) **Value of Count :** Bound at execution time.

(4) **Set of Possible Meanings for the Operator :** Bound at language definition time.

(5) **Meaning of the Operator-in this Statement :** Bound at compile time.

(6) **Internal Representation of 8 :** Bound at compiler design time.

3.4.1 Binding of Attributes to Variables

A binding is static if it occurs before runtime and remains unchanged (stable) throughout the program execution. If it occurs during runtime or can change in the course of program execution then it is called dynamic. The term stability refers to after binding, can attributes be modified or not. In static binding, attributes cannot be modified after binding where as in dynamic binding, the attributes can be modified.

3.4.2 Type Binding

Before a variable can be referenced in a program, it must bound to a data type. The type can be specified statically using explicit declaration like in C or implicit declaration like in FORTRAN. It can also be specified dynamically by associating a value during runtime like in APL.

3.4.2.1 Static Type Binding

An explicit declaration specifies a type of a variable using the data types available in a programming language. An implicit declaration is a method of associating variables with types through default conventions instead of declaration statements. In this case, the first appearance of a variable name in a program constitutes its implicit declaration. Both explicit and implicit declarations create static bindings to types.

Languages designed before late 1960s namely Fortran, PL/I and Basic have implicit declarations. In Fortran, if a variable is not explicitly declared then its type is bound implicitly. If the variable begins with one of the letters in I, J, K, L, M or N or their lowercase versions then it is implicitly assumed as Integer type, otherwise it is assumed to be type Real.

Many of the languages designed after mid 1960s require explicit declaration of all variables except Perl and ML(Metal Language). Even though implicit declarations are convenient for the programmer, it reduces reliability because the typographical errors cannot be detected.

Example : In Fortran, after declaring a variable Alpha as an Integer, if a statement is written as Alpha = 5.6 to assign a value to Alpha, this error is not detected. Instead a new variable Alpa is implicitly declared with Real type.

Some languages use special characters in the beginning of a variable to indicate their type. For example in Perl, any name that begins with a $ is a scalar which can store either a string or a numeric value. If a name beings with @ it is an array. If it being with a %, it is hash structure.

In C and C++, declarations and definitions of variables may be different.

Example : A prototype of a function declares the return type and the data types of parameters. The function definition can occur at a latter stage of the program.

Advantages and Disadvantages of Implicit Declaration : When a programming language support implicit declarations, it has the following advantages and disadvantages.

Advantages

(1) Implicit declarations allows static type binding i.e., binding occurs before runtime and continues till the entire program execution.

(2) It reduces the overall burden of the programmers.

(3) Implicit heap-dynamic variables provides high degree of flexibility by writing efficient generic code.

(4) No runtime overhead is incurred during allocation and deallocation of static variables.

(5) It creates different namespaces for different type of variables by using the available special characters.

Disadvantages

(1) Implicit declarations may sometimes leads to errors that are difficult to identify.

(2) It affects reliability because the compilation process is prohibited from detecting any severe errors.

(3) Implicit heap-dynamic variables causes runtime overhead in maintaining the dynamic attributes, such as array subscript types and ranges.

(4) Implicit heap-dynamic variables are ignored by the compiler resulting in the loss of error detection capability.

(5) Automatic conversion of implicit declarations to default types may causes changes in a program hence creating problems for the programmers.

3.4.2.2 Dynamic Type Binding

In dynamic type binding, the type is not specified in the declaration statement. Instead, the variable is bound a type when it is assigned a value in an assignment statement. When the assignment statement is executed, the variable being assigned is bound to the type of the value, variable or expression on the right side of the assignment.

Dynamic typing provides greater flexibility to the programmer. Generic programs can be easily written which can deal with any type of input data. Usually the dynamically typed languages are interpreted.

In APL and Snobol-4 the binding of a variable to a type is dynamic. For example in APL,

After executing I ← 92

I is bound to type integer,

After executing I ← 17.1 2.5 35.0

I is bound to a float array with 3 elements.

There are few disadvantages with dynamic typing. They are,

(1) The error detection capability of compiler is missing. When a wrong type is assigned, it cannot be detected in dynamically typed languages. Instead, the wrong type is assigned to a variable. This disadvantage is also prevailing in some of the statically typed languages such as Fortran, C and C++ in which RHS is converted to the type of the LHS automatically.

(2) The cost of implementing dynamic attribute binding is high. As the type checking is done during runtime, a runtime descriptor is associated with every variable.

Relationship Between Dynamic Type Binding and Implicit Heap-Dynamic Variables :
In dynamic binding, a declaration statement will not specify the type of the variable being declared. It is only when the variable is assigned value (i.e., at run-time), the type is bound to the variable. The type bound to the variable is the type of value or expression that is being assigned to the variable.

Example : Consider the declaration in Visual Basic DIM var.

In the declaration statement above, variable 'var' is not bound any type. We bound the type to it during execution of assignment statement. The assignment given below binds integer type of variable 'var',

```
var = 123
```

Similar is the case with implicit heap-dynamic variables. These variables are bounded to heap-storage on assigning values to them, i.e., at run-time. Whenever new values are assigned to them, all their attributes are bounded. Implicit heap dynamic variables can be used to assign any value during run-time.

Therefore, the relationship between dynamic binding and implicit heap dynamic variables is that, in both of them binding occurs when an assignment statement is executed (i.e., at run-time). However, the former binds only the type but latter binds all attributes.

3.4.2.3 Type Inference

A recent language ML supports both functional and imperative programming. It uses type inference mechanism to determine the type of an expression without requiring the programmer to specify the types of variables.

Example : The following is a function declaration,

```
fun area(r) = 3.14159 * r * r;
```

The types are inferred from the type of the constant as real. The function area takes a real argument and produces a real result. When a constant is not used in RHS a type can be mentioned for a variable as shown in the following example,

```
fun square(x) : int = x * x;
```

The function square takes an integer argument and evaluates an integer result.

3.4.3 Storage Bindings and Life Time

The memory for a variable is allocated from a pool of available memory. This process is called allocation. The process of taking back the assigned memory is called deallocation. The life time of a variable is the time during which the variable is bound to a specific memory location. The life time begins for a variable when it is bound to a memory location and ends when it is unbound from that memory cell.

Scalar variables categorized into four categories. They are,

(1) Static variables.

(2) Stack-dynamic variables.

(3) Explicit heap-dynamic variables.

(4) Implicit heap-dynamic variables.

3.4.3.1 Static Variables

Static variables bound to their memory cells before the program execution begins and remain bound until the program execution terminates. A globally accessible variable will be bound to its memory location until the program execution terminates. A static variable declared locally in a sub program is bound to its memory location even after that subprogram execution terminates. So, the value of a variable in the previous call to a subprogram can be tracked back. The addressing of static variables can be direct. No runtime overhead is required for allocation and deallocation of static variables.

Limitations

(1) Recursive subprograms cannot be supported.

(2) Storage cannot be shared among variables.

All variables in Fortran I, II and IV were static. In C, C++ and Java a programmer can use the static specifier to declare a static variable. Pascal does not provide static variables.

3.4.3.2 Stack-Dynamic Variables

Stack-dynamic variables are those whose storage bindings are created when their declaration statements are expanded. Storage is allocated when the execution reaches the declaration. So, the allocation is done at runtime.

Example : In Pascal the variables are declared in deallocation section. Whenever a code in a section is going to be executed, before that the memory for the variables is allocated at runtime on a runtime stack.

In C, C++ all local variables are stack-dynamic variables by default. In Pascal and Ada, all non-heap variables defined in sub-programs are stack-dynamic. All attributes other than storage is bound to the stack-dynamic scalar variables statically.

3.4.3.3 Explicit Heap-Dynamic Variables

Explicit heap-dynamic variables are nameless memory cells that are accessed using pointers. The allocation and deallocation of memory will be done using explicit runtime instructions. These are allocated on the heap and deallocated from the heap where heap is a garbage memory area which can grow unpredictably. The allocation can be done using an operator (like in Pascal and C++) and a subprogram (like in C). These are bound to a type at compile time (static) and to storage during runtime.

In C++, allocation is done using new operator and deallocation is done using delete operator.

Example

```
int *i;
i = new int;        //allocating integer memory
...
delete i;           //deallocation of memory
```

In Java, all objects are explicit heap-dynamic and are accessed through reference variables. Java has no explicit deallocation mechanism because automatic garbage collection is used.

Explicit heap-dynamic variables are used for creating dynamic structures such as linked lists, trees which grow and shrink during runtime. The limitation of these variables is cost of references, allocations and deallocations.

3.4.3.4 Implicit Heap-Dynamic Variables

Implicit heap-dynamic variables are bound to heap storage only when they are assigned values. All their attributes are bound every time they are assigned. These variables have highest degree of flexibility. The disadvantage is the overhead due to the maintenance of all the dynamic attributes like type of array subscripts and range. Another problem is complexity in runtime error detection.

Example : In APL one can write,

A → 10 5 3 2 7

A will become a single dimensional array with 5 elements regardless of its previous type. Again if we write,

A → 5.7

Now A will become a float variable.

3.5 TYPE CHECKING

Type-checking refers to the process of verifying whether all the operations are applied in a proper manner or not i.e., whether the arguments and data types of an operation are appropriate or not. If the applied operation is inappropriate then it specifies the occurrence of an error in a program. So, the purpose of type-checking is to prevent errors.

Example : W = X–Y* Z

The above expression includes three operations, assignment, subtraction and multiplication such that each operation must contain two arguments of proper type. If the subtraction operation is intended only for integer types and if 'X" is of character type then a type error is resulted and needs to be type-checked.

Two types of type-checking exists,

(1) **Dynamic Type-Checking :** It refers to the type-checking process which takes place during run-time, prior to the execution of a particular operation. For the variables that are dynamically bounded in a language, dynamic type-checking is needed. All the operations in a program are type-checked in a sequence and the type tags associated with each argument are verified and the appropriate tags are attached to each operation in such a way that the succeeding operations can use them. Hence, the additional code must be inserted in a program which takes extra time and space thereby affecting the overall runtime.

Advantages

(i) Dynamic type-checking provides flexibility in program designing.

(ii) It reduces the burden of associating data types from the progra ers.

Disadvantages

(i) Some execution paths in a program are not checked during runtime which results in type errors that are difficult to remove.

(ii) Additional storage is required for implementing type tags.

(iii) It is expensive.

(2) **Static Type-Checking :** Type-checking that occurs during compile time i.e., during the program translation is called 'static type-checking'. It is used for the statically bounded variables.

Advantage : Error detection at compile time is more beneficial than the run-time error detection because it incurs less cost.

Disadvantage : Programmer's flexibility is reduced through static type-checking as fewer schemes are allowed

3.5.1 Type Coercion

Type coercion refers to the process of converting a variable of one type-to a variable of another type by following the languages specific rules. This type of conversion occurs due to type mismatch between the actual argument type and the expected type of an operation.

Example : When an integer type variable is added to a floating type variable, the integ variable is automatically coerced to float-type and the floating point addition takes plac

Two types of coercions exists, they are as follows,

Widening conversion refers to the representation of a smaller-type value to a val of larger type without losing any information.

Example : Widening conversion is converting on 'int' type value to 'float'.

Narrowing conversion refers to the representation of a larger type value to a val of smaller type with some loss of information.

Example : The conversion of 'float' type value to an 'int' type value.

During the process of dynamic type-checking, whenever a type mismatch occurs coercion takes place at that point of execution whereas in case of static type checking an extra code is added to the program that allows easier type conversions during th execution process. For example, in C language if a type mismatch is detected by th compiler then it finds an appropriate conversion operation and inserts it into the compile program resulting in efficient program execution.

3.5.2 Type Error

During the process of type-checking, if an operation has inappropriate data type and arguments, a type error occurs.

Example

```
c = a * b
```

If the operator '*' is used only for real types and 'a' represents some character typ variable then an argument type error is generated. In C, if a function require a 'float type value and if an 'int' type is passed to it, then type error occurs as the parameter types are not checked by the compiler. The type errors are difficult to debug and remove With dynamic type-checking, some of the type errors can be notified while performing operations. Which makes the type-checking process difficult and produces type error whereas, in static type-checking all operations along with the possible execution paths are checked for the presence of any type errors thereby increasing the speed of a program.

3.5.3 Enforcing Type Checking in Various Programming Languages

Coercion refers to the process of converting a value of one type to a value of another type. These type conversions can be either narrowing or widening. A narrowing type conversion converts a variable's value of a larger data type to a value of smaller data type.

Example : Conversion of an int type value to a value of type byte in Java is called the narrowing conversion.

The term narrowing refers to those mappings where every domain type doesn't have a corresponding value in the range. A widening type conversion converts a variable's value of a smaller data type to a value of larger data type.

Example : Converting an int type value to a float type value in C is an example of widening type conversion.

Widening conversions are almost always safer than the narrowing type-conversions but the potential problem with a widening type-conversions is that it reduces accuracy.

In the language implementations, widening conversions of an int type to a float type eliminates some of the accuracy. For example, in some implementations, integers are stored in 32 bits which permits the storage of atleast nine decimal digits of precision. But in many cases, floating point values are also stored in 32 bits that allows only seven decimal digits of precision. Thus, conversion of integer to floating-point value can result in the loss of two digits of precision.

Type conversions can be either explicit or implicit. Explicit type conversions are also known as casts. A cast is usually written in a manner in which the desired type is kept inside the parentheses just prior to the expression conversion as shown below,

```
(int) a;
```

The reason for writing the type name inside the parentheses is that the C language contains many type names with two-words such as short int.

An implicit type-conversion occurs during the expression evaluation or while performing the assignment operation. This implicit type conversions may occur in several early programming languages such as Fortran which allows mixed-mode arithmetic.

One of the design issues involved in arithmetic expressions is whether an operator can have different types of operands associated with it. Languages which do not allow these mixed-mode expressions must define the rules for coercions.

3.6 STRONG TYPING

A programming language is said to be strongly typed if it allows all type of errors (occurred within a program) to be detected either at compile-time or at run-time, prior to the execution of a statement in which they actually occur. Whether a language is statically or dynamical typed, it doesn't prevent it from being strongly typed. Strong typing results in more reliable programs.

Example : Ada is not a pure strongly typed language because, it permits the programmers to violate the type-checking rules by sending a request to suspend the process of type-checking for some type conversions. This type of temporary suspension can be done when the predefined library function unchecked-conversion is used. This function takes a parameter variable whose type matches with the type of a function and returns a bit string that indicates the current value of that variable. These type of conversions are specifically used for user-defined strong allocations and deallocations. As no actual type-checking is preformed, task of obtaining some meaningful value is the sole responsibility of the programs.

ML is a strongly typed language consisting of the variables whose types are statically determined either from its declarations or from the type inferencing rules.

Though Java and C# are based on C++, they're regarded as strongly typed languages in a similar manner as Ada. Because of the use of non type-checked 'union' type, C and C++ are not considered to be as strongly typed languages. Fortran 95 is also a non strongly typed language because of the use of equivalence relationship between the variables.

Advantages

(1) It increases the execution efficiency by allowing the compiler to efficiently allocate memory and generates machine code that can effectively manipulates the data.

(2) The efficiency of translation will be improved by reducing the amount of code that needs to be compiled.

(3) Detection of errors at the early stages speeds up the development process.

(4) Provides better optimized code to the compiler.

(5) Determining the type doesn't incure any run-time penalty.

(6) It enhance the security and reliability of a program by minimizing the number of execution errors.

Disadvantages

(1) Some flexibility will be reduced.

(2) Strongly typed languages finds very much difficulty in defining collections of heterogenous objects.

(3) They limit the expressiveness.

(4) They scarcely results in highly ...amic/reflective systems.

(5) Learning of strongly typed lang...ges takes longer time.

3.7 TYPE COMPATIBILITY

Two types are said to be compatible if,

(1) They can be used together without any modifications.

(2) They can be substituted in place of one another without any alterations and they can be combined to form a composite type.

Designing the type compatibility rules of a language is important as it affects the design of the data types and the operations.

3.7.1 Name Type Compatibility

Two variables are said to be 'Name Compatible' if their definitions occur either in the same declaration or in the declarations that uses the same type name.

Advantage : Easy to implement.

Disadvantages

(1) It is highly restrictive.

(2) Another problem occurs when the structured type variables are passed through parameters in the subprograms.

3.7.2 Structure Type Compatibility

Two variables are said to be 'Structure Compatible' if their types have similar structures.

Advantage : It is more flexible than name type compatibility because the entire structures of two different types can be compared.

Disadvantages

(1) Difficult to implement.

(2) It doesn't allows to make differentiation between the types with the identical structure.

3.8 SCOPE

The scope of a variable refers to the range of statements in which the variables is visible. A variable is said to be visible in some statement if it is referenced by that statement.

The scope of a variable depends on the block in which it is declared. When a variable is declared inside some block, its scope is confined to that block. Such a variable is called as a 'local variable' and the variables which are declared outside the block are referred as "global variables".

Example

```
if (a[i] < a[j])
{
    int t;
    t = a [i];
    a[i] = a[j];
    a[j] = t;
}
```

In the above example, variable 't' is defined inside the 'if' block hence, its scope restricted to that 'if' block. The variable scope can be of two types.

(1) Static scope.

(2) Dynamic scope.

3.8.1 Static Scope

It is a method of binding names to nonlocal variables. It was introduced by ALGO 60, which has been copied by many subsequent imperative languages and many non imperative languages. Static scoping is well named because the scope of a variable ca be statically determined and it is very prior to execution.

There are two categories of static-scoped languages,

(1) In which subprograms can be nested, it means it creates nested static scopes.

(2) In which subprograms can be nested.

The C-based languages doesnot allow nested subprograms, but Ada, Javascript and PHP they allow the nested subprogram.

3.8.2 Blocks

It allows a section of code to have its own local variables whose scope is minimized those variables are typically stack dynamic so, they have their storage allocated when the section is entered and deallocated when the section is exited such a section of code is called a block.

3.8.3 Evaluation of Static Scoping

Static scoping provides a method of non-local access that works well in many situations. However it is not without its problems.

Consider the program whose structure is shown in the Fig. 3.8.1.

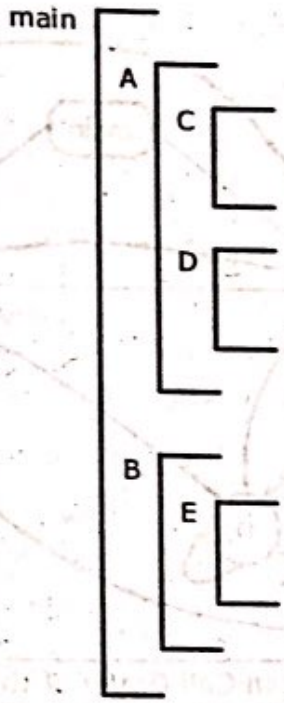


Fig. 3.8.1 The Structure of a Program

The above Fig. 3.8.1 shows this program contains an overall scope for main, section with two procedures that define scope inside main A and B. Inside A is the scope for the procedures C and D. Inside B is the scope of procedure E. To view the structure of a program as a tree it is very convenient in which each node represents a procedure and scope.

Fig. 3.8.2 shows the tree structure of a program and Fig. 3.8.3 shows the great deal of calling opportunity beyond that required in possible.

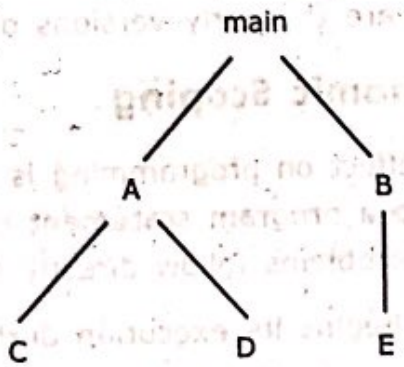


Fig. 3.8.3 The Tree Structure of the Program in Fig. 3.8.3

Fig. 3.8.3 shows the number of possible calls that are not necessary in this spec application.

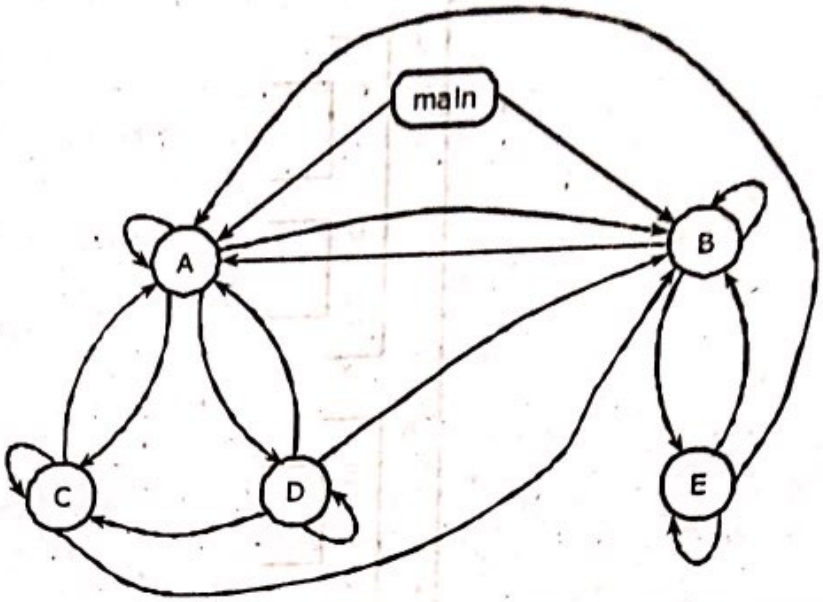


Fig. 3.8.3 The Potential Call Graph of the Program in Fig. 3.8.3

By mistakenly a programmer could call a subprogram that should not have bee callable, it not be detected as an error by the compiler. It delays detection of the erro until runtime, which make its correction more costly. Too much data access is also a mai problem.

Example : All variables in the main program are visible to all of the procedures whether or not that is desired and there is no way to avoid it.

The restrictions of static scoping can lead to program designs that bear little resemblance to the original, even in areas of the program even in changes have not been made.

3.8.4 Dynamic Scoping

Dynamic scoping is based on the calling sequence of subprograms not on their spatial relationship to each other. Then the scope can be determined only at run time. The scope of variables in APL, SNOBOL4, are the early versions of LISP in dynamic.

3.8.5 Evaluation of Dynamic Scoping

The dynamic scoping has effect on programming is profound. The correct attributes of non-local variables visible to a program statement cannot be determined statically. Several kinds of programming problems follow directly from dynamic scoping.

(1) First, when a subprogram begins its execution during the timespan beginning and ending when that execution ends, the local variables of the subprogram are visible to any other executing subprogram.

(2) Second, the problem with dynamic scoping makes program much difficult to read, because the calling sequence of subprograms must be known to the meaning of non-local variables.

(3) Dynamic scoping is inability to statically type check references to nonlocals.

(4) Accesses to non-local variables in dynamic-scoped languages take far longer than accesses to non-locals when static scoping is used.

3.9 SCOPE AND LIFETIME

The scope and lifetime of a variable sometimes it appear similar and related.

Example : Consider a variable that is declared in Java method that contains no method calls. The scope of such variable is from its declaration to the end of method the lifetime of that variable is the period of time beginning when the method is entered and ending when execution of the method terminates.

The scope and lifetime of the variable are clearly not the same, because static scope is a textual or spatial, concept whereas lifetime is a temporal concept.

Example : C and C++, a variable that is declared in a function using the specifier static is statically bound to the scope of that function and is also statically bound to storage.

Scope and life time are also unrelated when subprogram calls are involved. Consider the following C++ functions,

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

The scope of the variable sum is completely contained within the compute function. It does not extend to the body of the function printheader, although printheader executes in the midst of the execution of compute. However, the lifetime of sum extends over the time during which printheader executes. Whatever storage location sum is bound to before the call to printheader, that binding will continue during and after the execution of printheader.

3.10 REFERENCING ENVIRONMENTS

The referencing environment of a statement is a collection of all variables that a visible in the statement. These statement in a static scoped language is the variabl declared in its local scope plus the collection of all variables of its ancestor scopes th are visible.

The referencing environment of a statement includes the local variables, plus all the variables declared in the procedures in which statement is nested. Each procedu definition creates a new scope and thus a new environment. Consider the following A program,

```
procedure Example is
A, B : Integer;
   ...
   procedure Sub1 is
      X, Y : Integer;
      begin -- of Sub1
      ... <----------------------1
      end; -- of Sub1
   procedure 'Sub2 is
      X : Integer;
      ...
      procedure Sub3 is
         X : Integer;
         begin -- of Sub3
         ... <-----------------2
         end; -- of Sub3
      begin -- of Sub2
      ... <----------------------3
      end; -- of Sub2
begin -- of Example
... <-------------------------------4
end -- of Example
```

The referencing environments of the above program points are as follows,

Point	Referencing Environment
1	X and Y of Sub1, A and B of Example
2	X of Sub3, (X of Sub2 is hidden), A and B of Example.
3	X of Sub2, A and B of Example.

Consider the above program, firstly the scope of sub1 is at a higher level than sub3, the scope of sub1 is not a static ancestor of sub3, so sub3 doesnot have access the variables declared in sub1. There is a reason for this because the variables declared in sub1 are stack dynamic, so they are not bound to storage if sub1 is not in execution. Because sub3 can be in execution when sub1 is not, it cannot be allowed to access variables in sub1, which not necessary be bound to storage during the execution of sub3.

A subprogram is active if its execution has begun but not yet terminated. The referencing environment of a statement in a dynamically scoped language is the locally declared variables and the variables of all other subprograms that are currently active.

Assume that the only function calls are the following, main calls sub 2, which call sub1 considering an example,

```
void sub1( ) {
    int a, b;
    ... <------------------1
}   /* end of sub1 */
void sub2( ) {
    int b, c;
    ... <------------------2
    sub1;
}/ * end of sub2 */
void main( ) {
    int c, d;
    ... <------------------3
    sub2( );
}   /* end of main */
```

The referencing environments of the above program points are as follows,

Point	Referencing Environment
1	a and b of sub1, c of sub2, d of main, , (c of main and b of sub2 are hidden)
2	b and c of sub2, d of main, (c of main is hidden)
3	c and d of main

3.11 NAMED CONSTANTS

A named constant is a variable that is bound to a value at the time when it is bo to storage. Its value cannot be changed by assignment or by an input statement. Na constants aid in improving readability of a program.

When a constant value is to be used at various points of a program, it is be to use a name constant because when this value need to be modified it can be modi only at one place.

Example : In Pascal,

```
program test;
const s = 50;
```

The constant s can be used at various points of the program.

SYLLABUS [B.E. - OU]

Primitive Data Types, Character String Types, User-Defined Ordinal Types, Array Types, Associative Arrays, Record Types, Union Types, Pointer and Reference Types.

Algol-60	Integer, Real, Boolean
Pascal	Integer, Real, Boolean, Char
Ada	Integer, Float, Boolean, Character

A data type is said to be scalar if all the values in its domain are constant. Example of scalar types are 'integer, 'real', 'boolean' and 'char'. Another type called the 'structured type' also exists whose domain consists of members which are themselves composed of a set of types i.e., the structured type elements have fields which have their own types. Example of structured data types are 'arrays' and 'records'.

Importance of Data Types

(1) If data types are not supported, the compiler can't identify the object to which the name refers. Determining the object type also causes difficulty for the compiler.

(2) In case if no data types exists there would be no type checking which produces adverse effects on debugging and error checking. It also effects the program's reliability.

(3) Absence of data types also affects the languages's readability as the objects with distinct characteristics would not be clearly isolated.

4.2 PRIMITIVE DATA TYPES

All programming languages provides a set of primitive data types A primitive value is a value which cannot be decomposed into simplex values. Each and every language provides built in primitive types. Some languages also allows to define new primitive types. The primitive data types of language are used, along with one or more constructors, to provide the structured types.

4.2.1 Numeric Types

Many early programming languages provides only numeric primitive types. The following are the various types of numeric primitive data types.

4.2.1.1 Integer

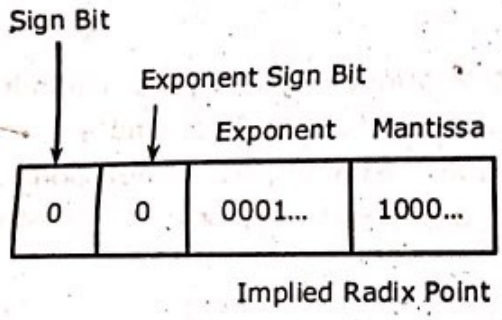
The most common primitive data type is integer. The length of the integer number in bytes can vary depending on the machine architecture. To stand up with these variations, languages provide various types of integers such as short, integer, long integer. An integer is a number without any decimal value. It can be either positive or negative. The negative integers can be stored in the form of signed magnitude representation or 1's complement or 2's complement representation. In most of the machines 2's complement is used to represent negative integers. Some examples of valid integers are -12, +25, +7.

C language supports short, int and long integers. The lengths of these data types are implementation dependent. In most of the machines short and integer are of two bytes length and long is of 4 bytes length. For representing the integer values ranging between -32768 to +32767 two bytes integer can be used. Some times qualifier such as unsigned can be used before integer to represent only positive values.

4.2.1.2 Floating-Point

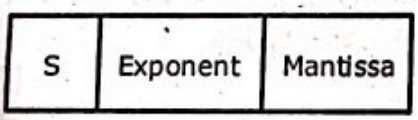
A floating-point data type can be specified with a data type real in FORTRAN, float in C (or) real in Pascal. The values for the reals are not orderly sequenced in a range of numbers. The precision required for floating-point numbers in terms of number of digits may be specified by the programmer as in Ada. The same arithmetic, relational and assignment operations described for integers are also provided for reals. The boolean operations are restricted slightly. Equality between two real numbers is difficult to achieve. Programs that check equality to exit a loop may never terminate. For this reason, equality between two real numbers may be prohibited by the language designer.

Floating-point numbers are stored in the form of scientific notation. The storage will be divided into a mantissa and an exponent as shown in the Fig. 4.2.1(a). It assumes that any number N can be represented as $N = m \times 2^k$ for m between 0 and 1 and for some integer k.



Sign bit = 0 implies positive number

(a) REAL (Floating Point)



S = one bit sign field,
E = an exponent in excess = 127 notation. Values range from 0 to 255 corresponding to exponents of 2 that ranges from –127 to 128,
M = Mantissa of 23 bits.

(b) 32 Bit Format of IEEE Standard 754

Fig. 4.2.1 Floating-Point Number Representation

IEEE standard 754 became the accepted standard for floating-point number representation. It specifies 32 and 64 bit standard for floating-point numbers. The 32 bit standard is as shown in 4.2.1(b). In 32 bit format, the range of numbers are 10^{-38} to 10^{+38}. In 64 bit format, the range of numbers are 10^{-308} to 10^{+308}.

A double precision floating point number will take 8 bytes and more precision than single precision number (32 bit floating-point number). Both single and double precision numbers are supported by addition, subtraction, multiplication and division operations.

C supports single precision and double precision numbers called float and double respectively.

4.2.1.3 Complex

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

Example

Fortran, Python.

In Python. the imaginary part of a complex literal is specified by following it with a J or j.

Example

(3 + 5j), (8 + 9J).

4.2.1.4 Decimal

Decimal data types store decimal numbers with fixed number of decimal places. They are useful in situations where large decimal values are needed but exact precision is important. Most business data processing require decimal data types. So, they are essential for languages like Cobol. The format of storage of decimal data types is given below,

Syntax

{Decimal|Dec} [(precision,[scale])]

The precision (the total number of digits before and after the decimal point) must be in the range between 1 and 31. The scale (the digits after the decimal point) must be less than or equal to the precision.

When precision and scale are not specified, the default settings are 5 and 0 respectively.

In a Decimal(P, S) the largest absolute value stored without errors is $10^{P-S} - 10^{S}$.

Example

Decimal(18, 9)

In the above example, 18 represents precision and 9 represents scale. It has nine digits on each side of the decimal point.

Advantages

(1) The advantage of decimal data type over floating-point data type is that they can store decimal values accurately within a limited range.

(2) According to the application, we can set the precision accurately.

(3) Decimal data types represent numbers upto a maximum of 32-digits precisely.

Disadvantages

(1) Decimal variable takes 16 bytes of memory whereas, floating variable takes only 8 bytes of memory.

(2) Decimal operations are slower than floating point operations.

(3) Decimal data types do not support exponents, so the range of values is limited.

4.2.2 Boolean Types

The boolean data type has only two values in its range. The value can be either true or false. In Pascal and Ada, the boolean data type is defined as an enumeration.

```
type Boolean = {false, true};
```

The ordering of the values is false<true. In most of the languages boolean values are restricted for input/output operations. Boolean types were introduced in Algol-60 and have been included in most of the languages. In C language, numeric expressions can be used as conditions. In such expressions, all the operands with nonzero values are considered true and zero is considered as false. Java has an explicit boolean data type.

Boolean types are used to represent switches or flags in programs. Integers can also be used for this, but boolean types are more readable. Boolean types can be implemented using one bit, but it cannot be accessed efficiently on many machines, so a byte memory is used. A zero in the entire byte represents false and any other bit string represents a true value.

4.2.3 Character Types

Characters are stored in computers as numeric codings. The most commonly used coding is 8-bit American Standard Code for Information Interchange (ASCII), which uses the value between 0....127 to code 128 different characters. In the 8-bit extended ASCII, the values 0....255 are used, which encode 256 characters. Most of the languages are providing a character data type. The characters include digits, alphabets and special characters.

The ordering of the character set is called the collating sequence for the character set. The collating sequence is used in string comparison. In the recent years, Unicode character set is gaining popularity because of international business of software. It uses 6 bits for representing a character. The first 256 characters are same as ASCII. It includes me of the languages other than English. Java uses Unicode character set. C uses ASCII racter set.

4.3 CHARACTER STRING TYPES

A character string type refers to a data type in which all the values comprises of a sequence of characters. Constants of this type are oftenly used for labeling all kinds of input and output data. These types are required by all the programs that involve character manipulation.

4.3.1 Design Issues

Following design issues occur in character string types,

(1) Is it necessary for the strings to be a special type of character array or any primitive data type?

(2) Is it necessary for the strings to have static or dynamic length?

4.3.2 String and their Operations

If a data type "string" is not available as a primitive data type in any programming language, then it can be stored in the single character array and can be referenced in a manner permitted by that language.

Usually, the languages such as, C and C++ uses 'char' type arrays for storing character strings and provide a standard library of all the string operations under the header file named as string.h. Most of the library functions follow a convention, that character strings must be terminated with a special character called 'NULL', represented as zero (0).

The commonly used string functions in C and C++ are,

(1) **strcpy :** It copies the strings.

(2) **strcat :** It concatenates one string at the end of another string.

(3) **strcmp :** It compares two strings in lexicographic order.

(4) **strlen :** It returns the length of a string. That is, the number of characters excluding the null character.

Java provides the String and StringBuffer classes for constant and variable length strings. Fortran 95 provides a substring reference operation for string handling.

4.3.3 String Length Options

The following options are available for determining the length of a string,

(1) Static length options.

(2) Dynamic length options.

4.3.3.1 Static Length Options

(1) The length of a string can be fixed (static) and is set at the time of its creation, such type of string is called as static length string.

(2) The varying string length is allowed till its declaration and is fixed at its maximum value set by the variable's definition, such type of strings are called limited dynamic length strings.

(3) The third option consists of a variable length string with no maximum value set. These type of strings are called dynamic length strings.

4.3.3.2 Dynamic Length Strings

These are the strings with varying length and no fixed maximum, set by the variables definition. These strings incur additional overhead of dynamic memory allocation and deallocation but, offer maximum flexibility. Also the length of the string increases or decreases, because the storage space to which it is bounded grows or shrinks dynamically.

The following two approaches are required by the dynamic length strings for their allocation and deallocation,

(1) The first approach is to store the dynamic length strings in a linked list such that, when a string grows, the new cells that are needed can be obtained from any location in the heap.

The disadvantage of using this approach is that the extra storage space is occupied by the linked list representation.

(2) The second approach is to use adjacent memory cells for storing complete strings. The problem arises, when the string grows. Hence, a new area of memory is used to store the new string completely. The old part of memory is moved to this new location thereby deallocating the memory cells used by the old string.

Though the linked list representation occupies large memory space, its allocation and deallocation is simple and some string operations are slow. On a contrary, the string operations are fast and require less space in case of adjacent memory representation but, the process of allocation and deallocation are slow.

4.3.4 Evaluation

String types are important to the writability of a language. Dealing with a primitive string is simple than dealing with strings as arrays.

Example : A language which treats strings as arrays of characters not have apredefined function as strcpy. Then this simple assignment of one string to another would require a loop.

Hence, it is difficult to justify without string primitive data types in languages. With string data types, the overhead of implementation must be reduced.

4.3.5 Implementation of Character String Types

The character string types are represented as character arrays. A static character string type descriptor, which is required only at the time of compilation has three fields.

Fig. 4.3.1 illustrates the compile time descriptor for static strings.

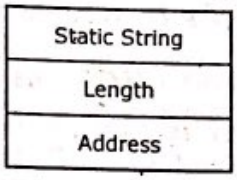


Fig. 4.3.1 Compile Time Descriptor for Static Strings

From the Fig. 4.3.1, the first field specifies the name of the type. The second field specifies that the types length (in characters). The third field is the address of the first character. A dynamic character string type descriptor only required at the time of Run-time. It stores both the fixed minimum length and current length.

Fig. 4.3.2 illustrates the run-time descriptor for limited dynamic strings.

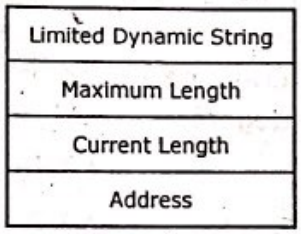


Fig. 4.3.2 Run-Time Descriptor for Limited Dynamic Strings

4.4 USER-DEFINED ORDINAL TYPES

An ordinal type refers to the type of data in which the range of different possible values can be connected to a set of positive integers.

Two kinds of user-defined ordinal types that are oftenly used in most of the programming languages includes,

(1) Enumeration types.

(2) Subrange types.

4.4.1 Enumeration Types

Enumeration types provides a method of defining and forming collections of named constants, called the enumeration constants.

The keyword enum lists out the words by implicitly assigning values 0, 1, 2,....., but it doesn't mean that, no explicit assignments of integer literals are possible with enumeration constants.

Example

```
enum months {Jan, Feb, Mar, Apr, May};
```

4.4.1.1 Design Issues

Various design issues associated with enumeration types are,

(1) *Is it possible to have coercion of any other type of an enumeration type?*

(2) *Does enumeration typed values coerced to integers?*

(3) *Is it possible to have an enumeration constant in more than one type definitions? And if so, how its occurrence is checked in a program?*

All the above mentioned design issues are associated with type-checking. In case of coercion of an enumeration variable to a numeric type, the control over its range of values is affected whereas when an 'int' type value is coerced to an enumeration type, any integer value irrespective of whether it is representing an enumeration constant or not, is assigned to an enumeration type variable.

Ada supports a technique called an overloaded literals, in which same enumeration literals can appear in more than one declaration within the same referencing environment. Which overloaded version of the literals to be used, is decided based on the type and context of the occurrence of a literal. The enumeration types in C# is similar to C++ except that they're not coercible to integers.

4.4.1.2 Evaluation

(1) Enumeration types provides both readability and reliability. In providing readability, all the named values are easily identifiable thereby making the coded values unrecognized.

(2) Enumeration types enhances the reliability of Ada and C# by offering the two advantages as follows,

(i) Applying any arithmetic operations an enumeration types is illegal.

(ii) Any value that lies outside the defined range can't be assigned to an enumeration variable.

As the enumeration variables are considered just as the integer variables in C, it does not offer the above mentioned advantages.

The efficiency of enumeration type variables increases as we move from C to C++ and C#, because C++ allows numeric value assignments to enumeration type variables and no coercion from enumeration type to integer type is allowed in C#.

4.4.2 Subrange Types

A subrange type is a type consisting of a contiguous subset of values that can be associated with a set of positive integers. Subranges in Ada are available in the subtypes category.

Example

```
type Days is (Day1, Day2, Day3, Day4);
subtype Weekdays is Days range Day1, ....,      Day4;
subtype Index is Integer range 1 ..... 100;
```

Subrange types causes the compiler to produce range-checking code for every assignment of a value to a subrange variable. As type checking takes place at the compile time, range checking must occur at run-time simultaneously.

User-defined data types allows the programmers to generate unique type for each distinct class of variables in the problem space.

4.4.2.1 Ada's Design

In Ada, subranges are included in the category of types are called subtypes. Subtypes are not new types but they are new names for possibly restricted (or) constrained, versions of existing types.

Considering an example.

```
type Days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
subtype Weekdays is Days range Mon..Fri;
subtype Index is Integer range 1..100;
```

From the above example, we can say that the restriction on the existing types is in the range of possible values. All the operations are defined for the parent type are also defined for the subtype except the assignment values they are specified at the outside range.

Example

```
Day1 : Days;
Day2 : Weekdays;
...
Day2 := Day1;
```

The assignment is legal unless the value of Day 1 is Sat. (or) Sun.

One of the most important type of user-defined ordinal types is for the indices of arrays. They can also be used for loop variables. The subrange types are very different fro ADs's derived types.

Considering the following example form type declarations.

```
type Derived_Small_Int is new Integer range 1..100;
subtype Subrange_Small_Int is Integer range 1..100;
```

Derived_Small_Int and Subrange_Small_Int, of the variables both types, have the same range of legal values and both inherit the operations of Integer.

4.4.2.2 Evaluation

(1) They enhances the readability as each user-defined type takes some meaningful name.

(2) All the variables are type-checked in order to ensure that the type compatibility rules have been followed.

(3) It increases modifiability because it allows the programs to change the type of variables by simply changing the type declaration statements.

4.4.3 Implementation of User-Defined Ordinal Types

Enumeration types are implemented as integers. With restrictions on ranges of values and operation. This provides increase in reliability.

Subrange types are implemented in exactly same way. Except that range checks must be included by the compiler.

4.5 ARRAY TYPES

An array is a collection of similar elements. These elements can be of any predefined data types such as int, char, float, double, etc. Each array element is identified according to its position with respect to the first element in an array. All the array elements must be of the same type and this type is called the base type of an array. The array variables are known as the indexed or subscripted variables.

4.5.1 Design Issues

Various design issues associated with arrays are as follows,

(1) *What data types are valid for array subscripts?*

(2) *When the ranges for subscripts be bounded?*

(3) *When allocation for an array occurs?*

(4) *What type of slices are permitted for an array?*

(5) *Which among the ragged or rectangular multidimensional arrays are allowed?*

(6) *Is it possible to initialize arrays when they have their storage allocated?*

(7) *Are the indexing expressions in element references, range checked?*

4.5.2 Arrays and Indices

An element of an array is referred using the name of the array followed by a variable known as subscript (or) index. If the indexes in a reference are constants, the selector is static, otherwise it is dynamic. The selection operation is a mapping from the array name and set of index values to an element in the array. Thus, arrays follow the principle of finite mapping. The mapping can be shown as;

array_name(index) → element

A problem with using parenthesis is, it can be confused as a function call. In languages like Fortran and PL/I, the name of the array is verified in the list of array names. If it is found, it is treated as an array reference otherwise a function call.

Example

sum = sum + A(I)

Fortran and PL/I use parenthesis for array subscripts, Ada also uses parenthesis for array subscripting. Pascal, C, C++ and Modula-2 and Java use square brackets to delimit their array indices.

The type of the subscript will be usually an integer subrange. Some languages like Pascal, Modula-2 and Ada allow other types (such as boolean, character and enumeration) to be used as subscripts.

In languages like Ada, Pascal and Java, the subscript is verified for range errors. Other languages do not perform range checking for the subscript, C, C++ perl, and Fortran do not perform range checking.

In many languages the dimensions of the array are not limited, Fortran I allowed up to three dimensions. So, three subscripts can be used. Later versions of Fortran support up to seven dimensions.

Arrays in C can have only one subscript, but arrays can have arrays as elements thus supporting multidimensional arrays. For example in C,

```
int A [5][6];
```

The declaration creates an array A with five elements where each element is an array of six elements.

4.5.3 Subscript Bindings and Array Categories

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

Arrays are classified into five types based on the binding to index value ranges and the binding to storage. They are,

(1) **Static Arrays :** A static array is an array in which both the bindings i.e., binding to subscript value ranges and binding to storage are static and takes place prior to run time.

Advantages

(i) Static arrays are efficient.

(ii) No need to do allocation and deallocation dynamically.

Fixed Stack-dynamic Arrays : In this type of an array, the binding to subscript value ranges is static whereas the allocation is dynamic and occurs at declaration elaboration time.

Advantage : A fixed stack-dynamic array is space efficient.

(3) **Stack-dynamic Arrays :** Stack-dynamic arrays are the arrays in which both the subscript ranges binding and storage allocation bindings are dynamic i.e., occurs during run-time. Once bounded, they remain fixed till the lifetime of a variable.

Advantages

(i) They are more flexible than the static and fixed stack-dynamic arrays.

(ii) It is not necessary to know the size of an array till it is actually used.

(4) **Fixed Heap-dynamic Array :** A fixed heap-dynamic array is similar to fixed stack-dynamic array i.e., both the subscript ranges binding and storage bindings are dynamic but they are fixed after allocating the required storage.

The major differences between the fixed stack-dynamic and fixed heap-dynamic arrays are as follows,

(i) Both the bindings occur when the user program issues the requests for doing so, rather than occurring at the elaboration time.

(ii) Instead of allocating the storage on the stack, it allocates the storage on the heap.

(5) **Heap-dynamic Array :** In this type of arrays both the bindings are dynamic and can vary indefinite times during the lifetime of an array.

Advantage : They are flexible when compared to other types of arrays hence can grow or shrink at any time during the program execution as per the space variations.

4.5.4 Heterogeneous Arrays

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

In Perl, the elements of an array can be mixture of numbers, strings and references. JavaScript is a dynamically typed language. Any array element can be any type.

4.5.5 Array Initialization

Arrays is some languages can be initialized at the time of allocating storage to them.

In Fortran, arrays are initialized by using the data statement as follows,

```
Integer a(3),
Data a/ 1, 2, 3, /.
```

Initialization implies assigning the initial values to array elements during the creation of an array.

ANSI C and C++ allow initialization of array.

Example

```
int s [ ] = {1, 5, 6, 9};
```

Array s is initialized with four elements taking the values from brackets sequentially. The compiler automatically sets the length of the array.

Character strings in C and C++ are initialized as follows,

```
char str [ ] = "Welcome";
```

The length of the array str is set to eight because all strings are terminated with a null character. It is implicitly supplied by the system to the string constants.

Array of strings can also be initialized as follows,

```
char * name [ ] = {"John", "Mary", "David"};
```

Name is an array of pointers where pointers are pointing to the strings given in brackets correspondingly. name[0] is pointing to "John".

Pascal and Modul-2 do not allow array initialization in the declaration part of the program.

The array List is initialized to the values specified in slashes.

Ada provides two mechanisms for the initialization of arrays. They are,

(1) Listing the values in parenthesis.

Example

```
L : array (1..5) of Integer := (2, 5, 7, 9, 11);
```

All the elements of an array 'a' are initialized with the given values in the order of their appearance, i.e., the first array elements contains 2, second contains 4 and so on.

(2) Directly assigning the value to an index position using => operator.

Example

```
S : array (1..5) of Integer := (1 => 5, 4 => 6, others => 0);
```

First element is initialized to 5 and the fourth element is initialized to 6 and the remaining elements are initialized to zero. The collection of values which are mentioned inside parenthesis are called "aggregate values".

.6 Array Operations

An array operation is one that operates on an array. The common array operations assignment, catenation, comparison. The C language do not provide any array erations. Perl supports array assignments but does not support comparisons..

Ada allows array assignments, concatenation through ampersand (&). Python rovides array assignment and comparison operators.

Fortran 90 includes various operations. They basically operate on pairs of array elements. Assignment, arithmetic, relational and logic operators are overloaded for array of any size. It also includes library functions for matrix multiplication, matrix transpose and vector dot product.

Example

C = A + B

A valid expression for array. The elements of A and B are added pairwise and assigned to the elements of the array C at the corresponding positions.

APL supports various operations on array. It is well known for its array manipulation. Four arithmetic operations are provided for matrices. It has unary operators for vectors and matrices. They are as follows,

Let V is a vector and M is a matrix,

(1) ⌽V reverses the elements of V.

(2) ⌽M reverses the columns of M.

(3) ⍉M used to find the transpose of a matrix 'M'

(4) ⊖M reverses the rows of matrix M.

(5) +M inverts 'M'.

Special operators that takes other operator as their operands are also provided by APL. An example of such an operator is the inner product operator, specified with a period (.).

It takes two binary operators as its operands as follows,

+.×

Let 'P' and 'Q' be the two vectors, the inner product of these two vectors can be obtained by multiplying their corresponding elements followed by their addition.

P+.×Q

The above statement is the sum of the inner product of 'P' and 'Q'. If 'P' and 'Q' are the matrices then this expression represents the multiplication of the two matrices 'P' and 'Q'.

4.5.7 Rectangular and Jagged Arrays

Rectangular Array : Rectangular array is a multidimensional array supported by certain programming languages like Fortran, Ada, C+#. In this array, the number of elements in a row/column is equal to the number of elements in the remaining rows/columns.

Rectangular arrays reduces the complexity by enabling the user to store multidimensional data with in single dimension array, i.e., the overhead incurred in creating arrays is decreased. These arrays even provides the user with the flexibility of using more than one index value with the same array.

In addition to this rectangular array helps in,

(1) Determining the size of array.

(2) Modelling the tables in an efficient manner.

Jagged Array : Jagged or rugged array is one-dimensional array supported by certain programming languages like C, C++ and Java. In this array, the number of elements in a row/column is not equal to the number of elements in the remaining rows/columns. Jagged array can also be defined as an array in which there exist a dimension that consists of one-dimensional arrays with different length.

Jagged array helps the programmer in creating,

(1) Array with various alternate dimensions

(2) Table in which the length of the rows are not equal

(3) Large 2-D arrays in which the elements are scattered.

Similar to rectangular arrays, jagged arrays also provides the flexibility of determining the size of array.

4.5.8 Slices

A slice of an array refers to some part (substructure) of the array. It is not actually a new data type, but specifies a technique for referencing some part of an array.

The syntax for providing a reference to some array slice is an important design issue to be considered while using array slicing. A reference to some element of an array consists of the array name and the subscript expression. As the slice of an array is actually a substructure of an array, a reference to it involves the use of fewer subscript expressions than are needed by the entire array. The missing subscript or the subscripts of slice references are denoted by asterisks.

Consider the following declarations in Fortran 95,

(1) Integer vector (3:6).

(2) Mat(1:4, 2).

(3) Cube(2, 1:3, 1:4).

first declaration vector (3:6) refers to the four element integer array specifying sixth elements of vector.

e second declaration indicates the second column of Mat.

ne third declaration specifies the array slicing in there dimensions.

The above described array slices are diagrammatically represented as shown in 4.5.1.

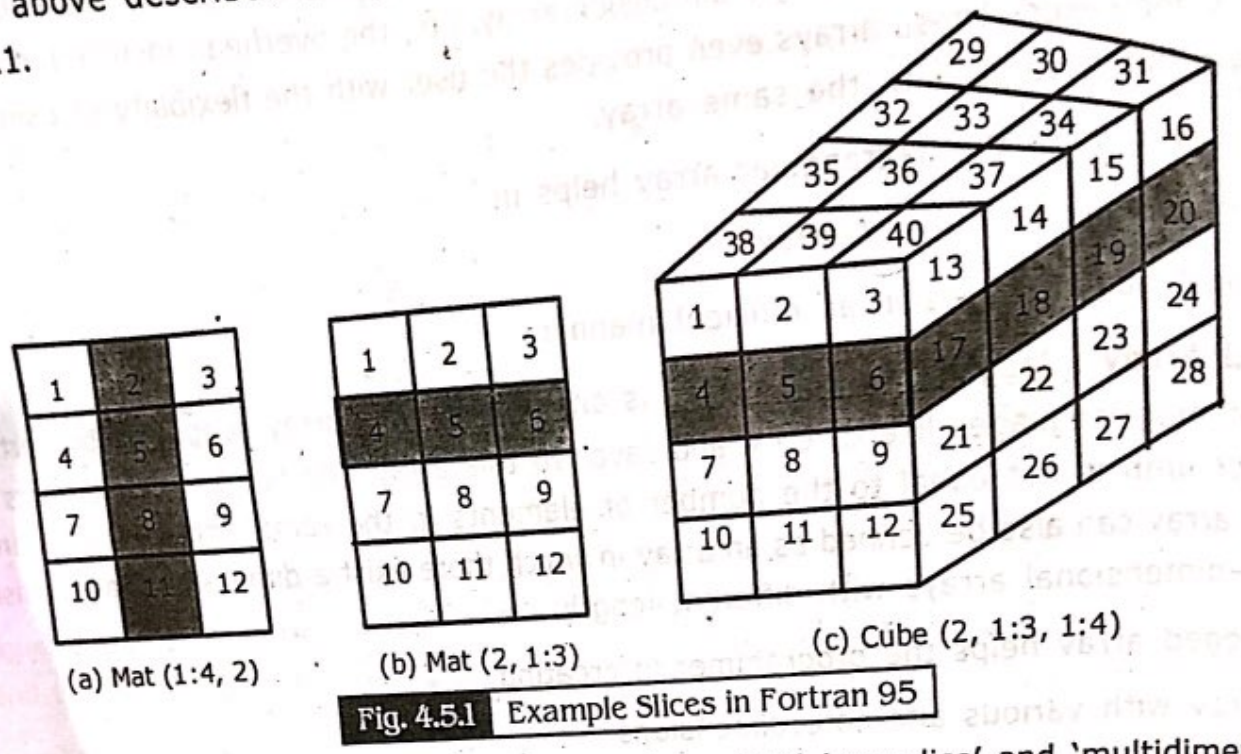


Fig. 4.5.1 Example Slices in Fortran 95

Fig. 4.5.1(a), (b) and (c) represents 'column slice', 'row slice' and 'multidimensional slice' respectively.

Ada permits only the highly restricted slices consisting of the consecutive elements of a single-dimensioned array.

4.5.9 Evaluation

Arrays have been included in all languages. They are simple and well developed. Although they are fundamental and essential, there is little controversy involved in their design.

4.5.10 Array Implementation

The code to allow accessing of array elements will be generated at compile time. At runtime, this code will be executed to produce element addresses.

Generalized Access Function for 1-D Array : In a single dimensional array, to find an element address the following method is used,

$$addr(A[k]) = addr(A[1]) + (k - 1) * elem\_size$$

where A[k] is the address of the element to be found, A[1] is the base address (beginning address) of the array A and (k – 1) * elem_size will give the size in bytes from the beginning of the array to $(k - 1)^{th}$ element.

For multidimensional arrays, the allocation of memory is sequential. But the elements of the array can be accessed using more than one subscript. There are two methods for mapping multidimensional arrays to single dimensional array,

(1) **Row Major Order :** The elements are mapped sequentially row wise.

(2) **Column Major Order :** The elements are mapped sequentially column wise.

Example

$$A = \begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{vmatrix}$$

Mapping using row major order results in 1, 2, 3, 4, 5, 6, 7, 8, 9.

Mapping using column order results in 1, 4, 7, 2, 5, 8, 3, 6, 9.

Location of A[i, j] in row major order is,

$loc(A[i, j]) = L_0 + (i - 1) * m * c + (j - 1) * c$

Where,

L_0 = Base address.

c = Number of locations allotted to a data type.

m = Number of columns.

Example : $loc(A[2, 1]) = 1030 + (2 - 1) * 3 * 2 + 0 = 1036$

So A[2, 1] beings at 1036

Location of A[i, j] in column major order is,

$loc(A[i, j]) = L_0 + (j - 1) * n * c + (i - 1) * c$

Where,

L_0 = Base address.

n = Number of rows.

c = Number of locations allotted to data type.

Example : The location of A[2, 1] in column major order is,

$$loc(A[2,1]) = 1030 + (1 - 1) * 3 * 2 + (2 - 1) * 2 = 1032$$

So A[2, 1] begins at 1032.

Fortran uses column major order and other languages like C use row major order.

Access Function for Multidimensional Array : The access function for multi-dimensional array is the mapping of base address of array and its index value to the address of element stored in memory. Generally,

address of an element = Base address of structure + (elem_size*numbers of elements preceding the array)

In row major order, the number of elements that precedes an element can be determined using the formula.

(number of rows*row_size above the elements) + (number of elements to the left of the element)

Therefore, actual address can be determined by multiplying the number of elements preceding the desired element by elements size.

add(a[p,q])=base_addr a[1; 1] + ((((number of rows above the pth row)*(row_size)) + (number of elements to the left of q^{th} column))*elem_size)

Where, m, n = row and column of the matrix.

Since number of rows above p^{th} row = (p–1) and number of elements to the left of the q^{th} column = (q–1) the above expression can be modified as,

addr(a[p,q])=base_add ra[1, 1] + ((((p–1)*n)+(q–1))*elem_size)

Where n = size of row (i.e., number of elements per row)

This can be rewritten as,

addr(a[p,q]=base_addr(a[1, 1] – ((n+1)*elem_size)+((p*n+q)*elem_size)

In this function, the first part specifies the constant part and the second part specifies the variable part. The generalized access function for arbitrary lower bound (lb) is,

addr(a[p, q] = base_addr[row_lb, col_lb] + (((p_row_lb)*n) + (q_col_lb)) * elem_size

Where,

row_lb = lower bound of rows

col_lb = lower bound of columns

The above generalized access function can be written as follows.

addr(a[p, q]) = base_addr[row_lb, col_lb] – (((row_lb*n) + col_lb)* elem_size)+ (((p*n)+q)*elem–size)

Where the first two terms in the function specifies the constant part and the last term specifies the variable part.

4.6 ASSOCIATIVE ARRAYS

Associative array refers to an unordered collection of data elements. These elements can be indexed by using 'keys'. All the user-defined keys in associative arrays must be stored in the structure in such a way that each element contains a (key, value) pair.

Associative arrays are supported by Python, Ruby and by the standard class libraries of Java, C++ and C#.

4.6.1 Structure and Operations

Associative arrays are usually implemented using perl's design technique. However they are also supported by Java and C++ standard libraries. In perl, associative arrays are called hashes because hash functions are used to store and retrieve the elements. Every hash variable in perl must be preceded with a percent sign (%).

Hashes can be set to literal values with the assignment as in,

```
%average = ("Shazia" => 83, "Zainab" => 80);
```

The above example creates a two-element associative arrays, of which the first elements are the 'keys' and the second elements are the 'values'.

The 'value' is accessed as follows,

```
$average{"Shazia"} = 83;
```

A new element is added in the following manner,

```
$average{"Neelima"} = 75;
```

An element can be removed using the delete operator, as follows,

```
delete $average{"Zainab"};
```

All the elements of the hash can be emptied by assigning an empty literal to it, as shown below,

```
@average = ( );
```

In order to determine whether an element is present in the hash or not, an operator called the 'exists' is used,

```
if(exists $average{"Uzma"});
```

Design Issues : Following are the design issues associated with the associative arrays,

(1) *How the elements are being referenced?*

(2) *Are the associative arrays, static or dynamic?*

4.6.2 Implementating Associative Arrays

Initially, during the implementation of associated arrays in perl, some fixed amount of space is allocated, later when it reaches some threshold level, its strong space is expanded which is a costly operation as it involves the use of a new hash function and rehashing of all the existing elements into the structure.

4.7 RECORD TYPES

A 'record' refers to collection of heterogeneous data elements in which the individual elements are determined by their names.

Example : An 'employee' record consists of the fields such as employee name, employee number, basic pay, designation etc. Among the above mentioned fields 'employee number' and 'basic pay' are of numeric types and 'employee name' and 'designation' are of character types.

The declaration of this record in 'pascal' appears as follows,

```
Emp-record : Record
BEGIN
    Ename : Char[1-15];
    Eno: Integer;
    Ebpay: Real;
    Edesig : Char[1-15];
END
```

'Record' data type in C is supported by using a concept called 'structures' as follows,

```
struct Emp
{
    Int eno;
    Char ename[15];
    Float ebpay;
    Char edesig[15];
};
```

The record fields in many programming languages such as C, C++, Ada and Pascal can be accessed by using the dot notation. For a record 'i', its basic pay field i.e., 'ebpay' can be accessed using Emp.ebpay[i].

4.7.1 Definitions of Records

The major difference between a record and an array is the, homogeneity of elements in arrays versus the possible heterogeneity of elements in records. One more important difference between arrays and records is that records in some languages are allowed to include unions.

The COBOL from of a record declaration, which is part of the data division of a COBOL program. Consider 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.
```

From the above example we can say that the EMPLOYEE-RECORD record consists of the EMPLOYEE-NAME record and the HOURLY-RATE field. The numerals 01, 02 and 05 they begins the lines of the record declaration are level numbers, which indicate by their relative values of the hierarchical structure of the record.

Ada uses a different syntax for records, rather than using the level numbers of COBOL, In an orthogonal way, the record structures are indicated by simply nesting record declarations inside record declarations. In Ada, record cannot be anonymous, they must be named types. Consider the following examples,

```
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;
```

From the above example, employee record, the employee name record would need to define first and then the employee record, name it as the type of its first field.

4.7.2 References to Record Files

References to the individual fields of records are syntactically specified by several different methods, two of which name the desired field and it enclosing records. COBOL field references have the form.

feild_name OF record_name_1 OF..... OF record_name_n.

Most of other languages use 'dot notation' for field references.

Examples : The following is a reference to the field Middle in the earlier Ada record example.

Employee_Record. Employee_Name. Middle

4.7.3 Operations on Records

Two basic operations that can be performed on records are,

(1) Assignment operation and

(2) Equality operation.

For an assignment operation, some value can be assigned to the record field.

For an equality operation we can compare one field value with another field value of some other record.

COBAL provides statement for moving records is MOVE CORRESPONDING. This statement is used to copy the fields of specified source record to the destination record. It is possible only if the destination record has a field with the same name.

Example

MOVE CORRESPONDING INPUT_RD TO OUTPUT_RD.

The above statement can copies the fields from the input record to output record.

4.7.4 Evaluation

Records are valuable data types in programming languages. Records are used when the collection of data values is heterogeneous and the different fields are not processed in the same way. And the fields of a record often need not be processed in a particular order.

4.7.5 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 as shown in Fig. 4.7.1. Run-time descriptors for record are unnecessary.

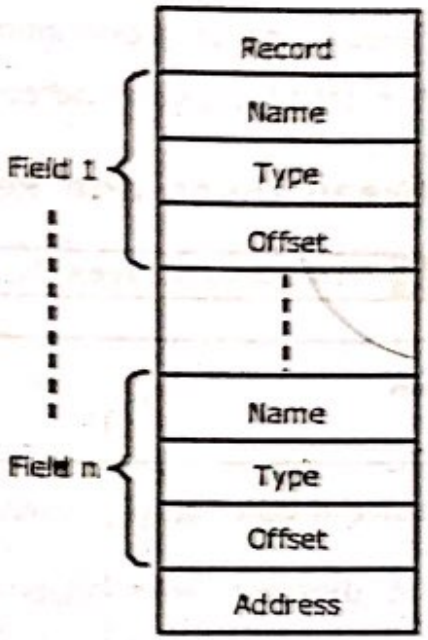


Fig. 4.7.1 A Compile-Time Descriptor for a Record

4.8 UNION TYPES

Union is a data type that may store values of different types at different times during the execution of a program. The memory required to store different types of values at different times is the same.

Example : Consider three types of constants i.e., 'int', 'float' and 'character'. These values can be stored one at a time in the table field. Thus, the type of such a location is the union of three types of values that can be stored in the table field.

4.8.1 Design Issues

The design issues for the union types are the following,

(1) Should type checking required?

(2) Should unions embedded in records?

3.2 Discriminated Vs Free Unions

Two types of union exists. They are,

1) **Free Unions :** Some languages like C and C++ provides the union constructs calle free unions because they offer sole freedom for type-checking to the programmers

2) **Discriminated Unions :** Algol-68 was the first language that supports 'Discriminate unions'. Later Pascal and Ada supported them. If type-checking for a union is to b done, a type indicator called a tag or a discriminant is required and a union wit this type of discriminant is called a discriminated union.

4.8.3 Comparison between Union and Record

Table 4.8.1 Differences between Union and Record

S.No.	Union	Record
(1)	Union is a data type that can store values of different types at different times during the execution of a program. The memory required to store different types of values at different times is the same.	A record is a heterogeneous collection of data elements whose individual elements are determined by their names.
(2)	Unions are used in programming languages such as C and Fortran. In Pascal and Ada, unions are implemented in the form of variant records.	Records are implemented in programming languages such as Cobol.
(3)	Discriminated unions uses the same address for every possible variant.	The record fields are stored in adjacent memory locations.

4.8.4 Comparison between Union and Enumeration

Table 4.8.2 Differences between Union and Enumeration

S.No.	Union	Enumeration
(1)	Union is a data type that can store values of different types at different times during the execution of a program.	Enumeration is a user defined data type, provides a method of defining an forming collections of named constants, called the enumeration constants.

(2)	**Syntax**
	union
	{
	Datatype member 1;
	Datatype member 2;
	⋮ ⋮
	Datatype member n;
	};
		Syntax
		enum datatype _name
		{
		value 1, value 2, ..., value n,
		};

(3)	The variables of union are declared as follows,
	union<tag>
	{
	Datatype member 1;
	Datatype member 2;
	⋮ ⋮
	Datatype member n;
	}
	variable 1, variable 2, ... variable n;
	The variable defined in union can be of different data types.
		The variables of enumeration can be declared as follows,
		enum datatype_name {var 1, var 2, ..., var n};
		The variable defined in enumeration type will have same data type.

(4)	Only one union member can be initialized at a time.	All variables of an enumeration can be initialized at the same time.
(5)	Unions are used in programming languages like C and Fortran.	Enumerations are used in programming languages such as, C, C++, C#, Pascal, Ada, Java.
(6)	In Pascal and Ada, unions are implemented in the form of variant records.	In Ada, enumerations are implemented in the form of overloaded literals.
(7)	Design issues that are associated with union types are, (i) Is it necessary to embedded unions in records? (ii) If it necessary to perform type checking is required?	Design Issues: Variables design issues associated with enumeration types are, (i) It is possible to have coercion of any other type of enumeration type. (ii) Does enumeration typed values coerced to integers. (iii) Is it possible to have an enumeration constant in more than one type definitions. And if so how its occurrence is checked in a program.

(8)	The variable define in union can be different data types.
	Example
	union organization
	{
	char name[25];
	char designation[10];
	int sal;
	};
		The variable define in enumeration type will have same data type.
		Example
		enum months
		{
		Jan, Feb, Mar, Apr, May
		};

4.8.5 Union Supported in 'C'

'C' supports union constructs but doesn't support type-checking.

Example

```
union example
{
    int a;
    float b;
};
```

4.8.6 Union Supported in Fortran

Fortran supports union in the form of 'Equivalence' statement. They do not suppor type-checking.

Example

```
integer a
logical b
real c
equivalence(a, b, c)
```

4.8.7 Union Supported Algol-68

Algol-68 was the first language to support 'discriminate unions'. In order to do type-checking for a union, a type indicator called a tag or discriminator is required.

Example

```
union(real, integer)x;
real a;
integer b;
case x in
(real n) : a : = n;
(integer m) : b : = m;
esac
```

4.8.8 Union Supported in Pascal

Pascal uses variant records in order to support 'discriminant unions'. It integrates unions with the record structure.

Example

```
type Shape = (Square, Circle, Triangle);
    Colors = (Cyan, Magenta, Red);
    Figure =
    record
    Color : Colors;
    Fill : Boolean;
    case Form : Shape of
    Square : (Side : integer);
    Circle : (Dia : real);
    Triangle : (Angle : real;
                Side 1 : integer;
                Side 2 : integer);
    end case ;
        varMyfig : Figure;
    end record;
```

4.8.9 Union Supported in Ada

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

4.8.10 Evaluation

Unions are unsafe data types in some languages. Such as Fortran C, and C++ which supports unions do not allow type checking. Unions must be used with care.

4.8.11 Implementation of Union Types

The implementation of discriminated union types is done by initially considering an address which can be associated the same address for with all the variants. Among all these variants, sufficient amount of memory must be allocated for largest variants. Whenever constrained variants are declared then every variant can be allocated with the same amount of storage space. This is because, these variants are reserved variable that doesn't support any sort of variants. Every variant in the union block has an associated discriminant, which is stored in a record like structure. Following Ada code illustrates the implementations of discriminated unions,

```
type compute_Node(Tag : Boolean) is record
case Tag is
when False ⇒ Calculate : Integer;
when True ⇒ Avg : Float;
end case;
end record;
```

A compile time descriptor is maintained for discriminated union types in order to store the description of every variant during the compilation process. The descriptor consists of the tag entry table associated with a case table.

The case table consists of an entry for every variant. This entry points to a descriptor. This can be illustrated by representing the diagrammatic form of compile time descriptor for a discriminated union.

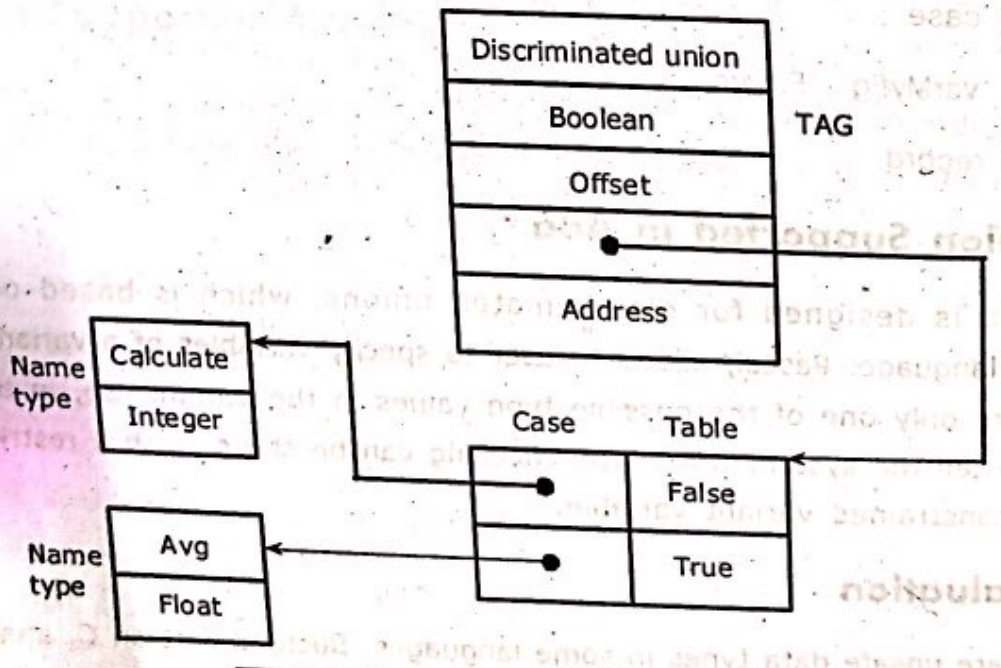


Fig. 4.8.1 Compile-Time Descriptor

4.9 POINTER AND REFERENCE TYPES

4.9.1 Pointer

A pointer is a reference to some object i.e., it is a variable that contains the address of some other variable. Using pointers, memory can be efficiently accessed allocated and released. Memory allocation is done dynamically.

Advantages

1) Pointers are used in indirect addressing which is most oftenly used in assembly languages.
2) Pointers are used in dynamic memory management (i.e., dynamic memory allocation and deallocation).
3) A pointer is used to access a location in the memory whose storage is allocated dynamically. This dynamically allocated storage is called a heap.
4) Variables that are dynamically allocated from the heap are called heap dynamic variables. The heap dynamic variables doesn't have the associated identifiers. Hence, can be referenced only by pointers and reference type variables.
5) Use of pointers makes the coding process simple and easy.
5) Pointers are used to create recursive types that are important in data structures and algorithms.
7) It enhances the languages writability.

4.9.2 Design Issues

The design issues for the pointers are the following,

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

4.9.3 Pointer Operation

Two basic operations are allowed on pointers when they are supported in the languages. They are,

(1) **Pointer Assignment :** Assigning a value to a pointer. Usually this value is an address of another variable. The address of the heap-dynamic variables are fetched using pointers, which can be used in other statements.

Example : Consider the following C++ statement,

```
a = *ptr;
```

In the above statement 'a' is a variable, 'ptr' is a pointer variable and '*' operator defines the pointer variable. Let 'ptr' contains a value '6586' which is the address for the value say '10' contained in 'a'. Thus, in the expression a = *ptr, 'a' contains the value '10'.

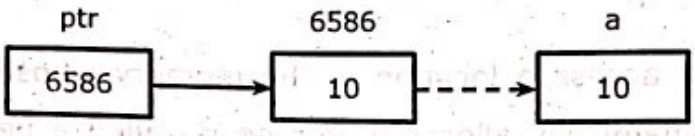


Fig. 4.9.1 The Assignment Operator a=*ptr

(2) **Dereferencing :** A pointer is considered as a reference, when it holds the address of other variable. A pointer can be used to access the value of a variable to which it is referencing, using indirect reference. It is called dereferencing of the pointer.

Example : In C, a pointer can be used in the following way,

```
int *p, x = 5;
p = &x;
```

A snapshot of memory allocation is as shown in Fig. 4.9.2.

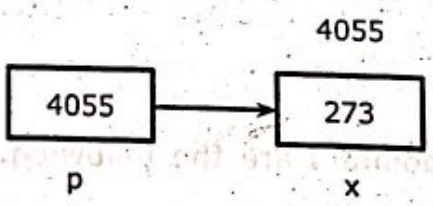


Fig. 4.9.2 Pointer Reference a Variable

A pointer p is holding the address of a variable x. p is used to access the value of x using dereferencing. It is referred as *p. x and *p are referring to the same location. Now, x can be modified using *p, 4055 is the address of x and 273 is the value of x.

Dereferencing can be either implicit or explicit. In Fortran-90 and Algol-68, it is implicit. In other languages it should be explicitly stated. For example in Pascal, ^ is used for dereferencing. In C, dereferencing is achieved by using *. When a pointer points to a record, a field can be referenced as (*p). Eno or p -> Eno where Eno is the filed of a record.

Insecurities of Pointers : PL/I was the first high-level language to introduce pointers. In PL/I pointers are untyped. They can point to the address of any memory location, so dynamic type checking is required to check the proper access of the content. Using pointers one can manipulate the lower level hardware configuration. To avoid the problems associated with pointers, Java replaced pointers with references.

4.9.4 Pointer Problems

The first high-level programming language to include pointer variable was PL/I, pointers can be used to refer both heap-dynamic variables and other program variables.

The pointers of PL/I were highly flexible, but their use lead to several kinds of programming errors.

Some languages such as Java, completely replaced pointers with reference types, along with the implicit deallocation, minimize the primary problems with pointers.

4.9.4.1 Dangling Pointers

A dangling pointer is a pointer that holds the address of a deallocated heap-dynamic variable.

They produce harmful effects as follows,

1) The location pointed to, by the dangling pointer may be reallocated to some other newly created heap-dynamic variable. If a type mismatch occur between the old and new variables, then the type checks made by using the dangling pointer becomes invalid.
2) In case, if a dangling pointer is applied to change the value of a heap-dynamic variable, the value currently holded by the new heap-dynamic variable get destroyed.
3) A change in the location leads to the failure of the storage manager.

A dangling pointer can be created in the following manner,

1) Initially, a pointer 'ptr1' is set such that, it points to the newly created heap-dynamic variable.
2) Another pointer 'ptr2' is assigned a value of 'ptr1'.
3) Finally, the heap-dynamic variable which is pointed to by 'ptr1' is deallocated in an explicit manner without changing 'ptr2'. Hence, 'ptr2' now acts as a dangling pointer.

4.9.4.2 Lost Heap-Dynamic Variables

A lost heap-dynamic variable refers to the allocated heap dynamic variable, which is no longer in use by any user program. They are also known as garbage, because they do not serve the purpose for which they are intended. Also they can't be reallocated to some other use by the program.

A lost heap-dynamic variable can be created in the following manner,

(1) A pointer 'ptr1' is set such that, it points to the newly created heap-dynamic variable.

(2) After some instant of time 'ptr1' is set to point to some other newly created heap-dynamic variable, hence the first heap-dynamic becomes inaccessible or get lost.

4.9.5 Support for Pointers Offered by Many Programming Languages

Pointers are supported by many programming languages like Ada, Pascal, C, C++, Algol-68 etc,

(1) **Pointers in C and C++ :** Use of pointer in C and C++ is similar to the use of addresses in assembly languages. They provide flexibility when used with great care. The problems of dangling pointer and heap-dynamic variables can't be solved using pointers, pointer arithmetic is also possible in C and C++. The pointers in C and C++ can virtually point to any variable located at any place in the memory.

Example

```
int *p;
int sum, elem;
.....
.....
p = &elem;
sum = *p;
```

In the above statements, asterisk (*) denotes the dereferencing operation and ampersand (&) is the operator for producing the address of a variable. An equivalent statement for the above two assignment statements is sum = elem.

Pointers can either be assigned with the address values of any objects (or) can be assigned with 'null' (constant zero).

Pointers in C and C++ can point to functions which allows passing functions as parameters to some other functions.

Pointers in C and C++ can be 'generic' of type void *, it means that they can point at any value of any type. As generic pointers can't be dereferenced, type-checking is not a problem.

(2) **Pointers in Fortran 95 :** Both heap-dynamic variables and static variables can be pointed by pointers in Fortran 95.

Example

```
Integer, Pointer : : Int_p
Integer, Pointer,Dimension (:): : Int_
```

Where, 'Int_p' can point to any integer type value and 'Int_List_p' can point at any single-dimensioned array containing integer type elements.

Implicit dereferencing is also possible with Fortran 95 pointers. They can also become dangling as the deallocate statement, which takes a pointer as its argument, can't determine in case if other pointers are pointing at a heap-dynamic variable that is being allocated.

(3) **Pointers in Ada :** Pointers in Ada are called access types. A dangling pointer which contains the address of a deallocated heap-dynamic variable causes serious effects in any programming language. The problem of dangling pointers is reduced to some extent by Ada. In Ada, a heap-dynamic variable might be implicitly deallocated at the end of the scope of its pointer type. This process lessens the need for explicit deallocation.

4.9.6 Reference Types

A 'reference type' variable in C++ refers to the special class of pointer (variables) which are basically used for the formal parameters in function definitions. As reference type variable is a constant, its initial value is the address of some variable that exists in its definition. Once it is initialized, it can't be set to refer to any other variable. The names of the reference type variables must be preceded with ampersand (&).

Uses : When they are used in function definitions, they allow two-way communication between the caller and the called functions. This can't be achieved using any non pointer primitive parameter types as all the parameters in C++ are passed through pass-by-value technique.

Another method to achieve two-way communication is to pass a pointer as a parameter but doing so, requires explicit dereferencing, thereby reducing the read-ability and safety of the code.

They are treated just like the other parameters in the called function whereas in a calling function they are not considered to be unusual things. When reference parameters are used as formal parameter, their addresses rather than their values are passed by the compiler.

4.9.7 Evaluation

Pointers are essential in some programming languages. Pointers are necessary to write device drivers, in which we specify absolute addresses must be accessed.

References of Java and C# provide some of flexibility and the capabilities of pointers, without hazards.

9.8 Implementation of Pointers and Reference Types

The implementation issues are as follows,

(1) **Representations of Pointers and References :** Based on size of the address space (or) a machine, pointers and references are stored as single values in either 2 or 4 bytes memory cells. This process is followed in many larger computers. But in most of the microcomputers in which segment and offset are regarded as two individual parts of an address, pointers and references are implemented as a pair of 16-bit words, one for each part of an address.

(2) **Solutions to the Dangling Pointer Problem :** A dangling pointer refers to a pointer that contains the address of a heap-dynamic variable that has been deallocated. Dangling pointers implementation leads to many dangerous effects. Many solutions have been proposed for the dangling pointers problem and one among them is 'Tombstones'. Every heap-dynamic variable consists of a special cell called a tombstone which itself is a pointer to heap-dynamic variable. The pointer variable doesn't points to the heap-dynamic variables but it points to the tombstone. When a heap-dynamic variable is deallocated, the associated tombstone exists but a reference to a pointer which is pointing to an empty tombstone is regarded as an error.

The two main drawbacks of tombstones are as follows,

(i) They are costly in time and space. The reason behind this is that the storage occupied by tombstones are never deallocated hence can't be reclaimed.

(ii) An additional level of indirection which involves extra machine cycle is needed for accessing a heap-dynamic variable through a tombstone.

Another approach to solve the problem of dangling pointers is the *'locks and keys'* approach. It is used in the implementation of UW-Pascal. In UW-Pascal, it acts as a compiler and the pointers are represented as ordered pairs consisting of 'keys' and 'addresses' i.e:, (key, address). Where, 'key' is an integer value. The general form representation for heap-dynamic variables is the sum of the storage occupied by that variable and a header cell that stores an integer lock value.

After allocating the heap-dynamic variable, a lock value gets created and is kept in the lock cell of heap-dynamic variable and key cell of the pointer. Whenever a dereferenced pointer is accessed, the key value is compared with the lock value. If they match, then their access is legal, otherwise a runtime error will occur. When a heap-dynamic variable is deallocated, its lock value is cleared to an illegal lock-value.

One best way to completely solve the problem of dangling pointers is to avoid the usage of deallocation of heap-dynamic variables by the programmers. If no explicit deallocation of heap-dynamic variables occur in programs then there will be no dangling pointers. Hence, they must be explicitly deallocated by the run-time system when they are not in use.

(3) **Heap Management :** Heap management is a complex run-time process. Two situations that can occur in heap management are as follows,

(i) **Single-size Cells :** In this type of allocation, the heap storage is allocated and deallocated in single size units. In this technique, a list of available space is formed and all the available cells are connected together using the pointers present in the cells. Memory allocation is a simple process but deallocation is a complicated process. Many pointers will be pointing to a heap-dynamic variable thus making the process of determining when the variable was unused, difficult. This led to the use of garbage.

There are two processes for reclaiming garbage. They are,

- **Reference Counters :** In reference counters, the process of reclamation is incremental and occurs during the creation of inaccessible cells. Each allocated block of memory i.e., a cell contains a counter, which stores the number of pointers pointing to the cell at any time. The counter value is incremented each time a new pointer starts pointing to the block and gets decremented when a pointer stops pointing to the block. When the count becomes '0', the block is returned to the free memory list. It is also known as the eager approach.

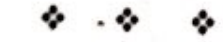
  Reference counters suffers from the following problems,

  - If the blocks are small, the storage taken up by the counters become crucial.
  - There is an execution time penalty in order to maintain the counter values because for every change in the pointer value, the cell previously pointed to by the pointer must decrement its value and the currently pointed cell must increment its counter value.
  - Circular cells connections causes much complications.

- **Garbage Collection :** In this process, the programmer requests allocation of memory from heap. If allocation is granted, the required memory is allocated and address is returned which is stored in a pointer variable. Contents of a pointer variable can be copied so that multiple pointers can point to the same memory location. The allocated area becomes garbage if it is no longer referenced by any pointer. It occurs when run-time system finds no free memory to be allocated.

(ii) **Variable-size Cells :** This approach of storage allocation suffers from all the problems that are associated with the single-sized cells in addition to the following problems,

- The initial setting of the cell indicators, to specify that they are garbage, is difficult. This problem can be solved by having the cell size as the first field of a cell, then they are scanned, this process of scanning requires more space and time.
- The location of pointers in the cells is important. We can add a system pointer to each cell, but this can be done by using user-defined pointers which requires additional space and execution time hence, increases the cost of executing a program.
- Maintaining a list of the available storage space also causes a problem. Initially all the available space must be treated as a single cell in a list. But if a list is too long consisting of many blocks, the process of allocation is slowed down as the request for a block causes the entire list to be searched. Thus, maintaining a list is an overhead.



R 5

EXPRESSIONS AND ASSIGNMENT STATEMENTS

S [B.E. - OU]

sions, Overloaded Operators, Type Conversions, Relational and Boolean Circuit Evaluation, Assignment Statements, Mixed-Mode Assignment.

5.1 INTRODUCTION TO EXPRESSIONS AND STATEMENTS

Expressions and statements are the fundamentals for specifying computations in a programming language. To understand the evaluation of expression, it is better to know the order of operators and operands evaluation. The order of operators is based on associativity and precedence rules of the language.

The purpose of an assignment statement is to change the value of a variable. This statement specify an expression to be evaluated and destination in which to place the result of expression.

5.2 ARITHMETIC EXPRESSIONS

In programming languages, an arithmetic expression is an expression that contains operators, operands, parentheses and function calls. The operators can be unary (operating on a single operand) or binary (operating on two operands) or ternary (operating on three operands). The ternary operators are available in C, C++ and Java.

The binary operators used in most of the imperative-programming languages are 'infix' i.e., they appear between their operands.

The implementation of arithmetic expressions results in two actions,

(1) Fetching the operands from memory.

(2) Performing the arithmetic operations on those operands.

5.2.1 Design Issues

The design issues for arithmetic expressions are as follows,

(1) What are the precedence rules for the operators?

(2) What are the associativity rules for the operators?

(3) What order is chosen for operand evaluation?

(4) Are there any impositions on operand evaluation side effects?

(5) What combination of types is permitted for the expressions?

(6) Is user-defined operator overloading allowed in the language?

2.2 Operator Evaluation Order

Following are the language rules that specify the order of evaluation of operators,

- Precedence.
- Associativity.
- Parentheses.
- Ruby expressions.
- Conditional expressions.

2.2.1 Precedence

When an expression includes many operations, each part or an expression is valuated and resolved in some predetermined order. This is called the *operator ecedence.*

The precedence of operators used in arithmetic expressions is based on the priorities signed to the operators. (Among the basic arithmetic operators, multiplication and vision have higher precedence over addition and subtraction operators).

The precedence rules of the common imperative languages specifies the order of valuation of the two adjacent operators having different precedences in an expression. these languages, the exponentiation operator has higher precedence than the four asic arithmetic operators.

Many programming languages contain unary addition and subtraction. Unary addition called the identity operator as it doesn't have an operation associated with it. In Java, nary plus and unary minus operators results in the implicit conversion of short or byte pe operand to an operand of type int.

A parenthesized unary minus operator can be placed either at the beginning or inside ny expression to prevent it from being adjacent to another operator.

xample : x + (–y) * z

The above expression is legal as unary minus is kept in the parentheses but the xpression below,

x + –y * c

This is illegal and decreases the readability of an expression. Hence, the precedence ules are applied to determine the order of applying the operators for evaluating the ub-expressions.

recedence Rules : The precedence rules of a language specifies the order of evaluation hen two operators with different precedences are adjacent to one another in an xpression.

Fortran	C-based Languages	ADA
* *	++, – – (Postfix)	* *, abs
*, /	++, – – (Prefix), +, – (Unary)	*, /, mod, rem
+, –	*, /, %	+, – (Unary)
+, – (binary)		+, – (binary)

The ** operator in the operators precedence table is an exponential operator used in exponentiation.

abs is an unary operator that returns the absolute value of its operand.

rem operator of Ada is similar to the % operator of C i.e., it accepts two operands and returns the remainder by dividing the first operand by the second.

mod operator is same as the rem operator when both the operands are positive.

5.2.2.2 Associativity

The associativity rules of a language specify which operator has to be evaluated first, when two operators having the same precedence appear adjacent to each other in an expression.

The associativity of an operator be either right associative or left associative, indicating that the evaluation begins at the left most or rightmost occurrences respectively. All the operators in common imperative languages are left to right associative except the exponentiation operator, which is right to left associative. The exponentiation operator in Fortran is right associative whereas it is non-associative in Ada, hence the expression x**y**z is legal and valid in Fortran whereas it is illegal in Ada and must be parenthesized to show the order of evaluation as follows,

(x * * y) * * z (or) x * * (y * * z)

The fact that some of the arithmetic operators are mathematically associative is often used by most of the compilers hence, does not cause any affect on the value of an expression by the associativity rules. The addition operator is mathematically associative hence, the value of an expression is independent of the order of evaluation. Simple optimizations can be performed by the compiler if associative floating-point operations are used. Faster code for an expression evaluation results from reordering the operators evaluation.

Associativity Rules : An operator can be either left, right or non-associative. The associativity rules of all the arithmetic operators in common programming languages are as shown in Table 5.2.2.

Table 5.2.2 Associativity Rules of Various Languages

Associativity	Fortran	C-based Languages	ADA
Left associative	*, /, +, –	*, /, %, +, – (binary)	All operators except * *
Right associative	* *	++, – –, –, + (unary)	
Non-associative			* *

5.2.3 Parentheses

The precedence rules for operators can be altered by the programmers by adding parentheses to the expressions. The precedence of a parenthesized part is more than its unparenthesized part.

Example : (x + y) * z

Multiplication operator has higher precedence over addition operator but, in the above mentioned example, addition has higher precedence over multiplication, this is due to the presence of parenthesis in the addition expression hence, addition operation is performed before multiplication operation. Languages that provides the facility of placing the parentheses in arithmetic expression can be evaluated either in left to right or right to left order.

Example : (x * (y + z))

In this expression, the evaluation starts from right to left because the second operand of the multiplication operator is not available till the addition operation in the parenthesized sub-expression is evaluated.

Thus, the programmers can specify the desired order of evaluation with parentheses and can change the precedence rules by placing the parentheses in expressions.

Advantage : The advantage of this method is that neither the writer nor the readers of programs need to be aware of any precedence or associativity rules.

Disadvantage : It makes the job of writing expressions more tedious and affects the readability of code.

5.2.4 Ruby Expressions

Ruby is a pure object-oriented language, it 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.

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.

5.2.5 Conditional Expressions

A conditional expression is a compound expression that contains three expressions, of which first expression is a condition, the second expression is evaluated when the condition is true and the third expression is evaluated if the condition fails.

In C language, the actions to be performed depends on the circumstances. Sometimes we execute one set of instructions or the another depending on the situations. This type of situations in C are called 'conditions'.

The conditional operators also called the ternary operators in C, are used to form conditional expressions. These are usually represented as '?' and ':'. When used in conditional expressions, they take the general form as follows,

```
expr_1 ? expr_2 : expr_3
```

The above expression specifies that "if expr_1" is true, then the value returned will be expr_2, else the value returned is expr_3.

Example

```
a = 5;
b = a > 2 ? 10 : 20;
```

The variable b has a value 10 if a > 2, else 20 is assigned to b. As a is initialized to 5 and since 5 > 2, the value assigned to b is 10.

The following points have to be noted about the conditional operators,

(1) Conditional operators can be used in any statements other than the arithmetic statements.

(2) Nesting of conditional operators is allowed in C.

(3) Conditional expressions can be used anywhere in C program where other expressions are used.

The limitation of conditional operators is that only one statement is allowed to occur after the ? or : operator.

5.2.3 Operand Evaluation Order

Another design characteristic of expressions is the order of evaluation of operands. 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 expressions, then all 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.

5.2.3.1 Side Effects

A programming language construct is said to have a side effect if it affects the subsequent computation which ultimately affects the output of a program. A functional language doesn't produce any side effect. Hence, the value of an expression depends on the refreshing environment in which an expression is evaluated but not on the time at which the evaluation occurs.

A side effect of a function (also known as a functional side effect) is a one that results from changing one of the function parameters or a global variable by a function. Consider the following expression,

```
x + function(x)
```

In case, if function doesn't produces any side effect, due to the changes made to x, the evaluation order of the two operands x and function(x) will not cause any significant effect on the value of an expression but if function changes x, it produces an effect.

Let the value of x be 6, then we define a variable y as follows,

```
x = 6;
y = x + function(x);
```

As function(x) returns a value, which is obtained by dividing its argument by 2 and changing its parameter value to 20. In evaluating the above expression, if the value of x is fetched first then the resultant value of an expression is 9.

In case, if the second operand is evaluated before x, then the value of the first operand is 20 and the resultant value of an expression is 23.

The problem of operand evaluation can be solved in two ways,

(1) By prohibiting function evaluation from affecting the value of an expression. This can be done by preventing any functional side effects.

(2) Another method of avoiding the problem of functional side effects is to specify the evaluation order for the expression operands in the language definition itself.

Hence, prohibiting functional side effects is a difficult process as it reduces the programmer's flexibility.

5.3.2 Referential Transparency and Side Effects

The 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 referentially transparent function value depends entirely on its parameters.

Following example gives the connection of referential transparency and functional side effects.

```
result1 = (fun(a) + b) / (fun(a) - c);
temp = fun(a);
result2 = (temp + b) / (temp - c);
```

From the above example we can state that the function fun has no side effects, result 1 and result 2 will be equal because the expressions assigned to them are equivalent. Suppose fun has the side effect of adding 1 to either b or c. The result 1 would not be equal to result 2. So that, the side effect violates the referential transparency of the program in which the code appears.

Advantages : The referential transparent program has the following advantages,

(1) The semantics of such programs in much easier to understand the semantics of programs that are not referentially transparent.

(2) In terms of easily understand referentially transparent makes a function equivalent to a mathematical function.

5.2.4 Solved Problems

SOLVED PROBLEM 1

Let the C function 'fun' be defined as,

```
int fun (int *k)
{
    *k + = 4;
    return 3 * (* k) - 1;
}
```

Suppose 'fun' is used in a program as follows,

```
Void main( )
{
```

```
    int i = 10, j = 10, sum1, sum2;
    sum1 = (i / 2) + fun (& i);
    sum2 = fun(& j) + (j / 2);
}
```

Determine the values of sum1 and sum2 by running the program on a computer. Explain the result.

SOLUTION

As the function fun produces the side effect of increasing the value of its argument by 4, the order of evaluation determines the result. If the left to right order is followed during the evaluation then sum1 results in 46 and sum2 results in 48 i.e.,

$$sum1 = \left(\frac{10}{2}\right) + 41 = 46 \quad \text{and} \quad sum2 = 41 + \left(\frac{14}{2}\right) = 48$$

If the evaluation starts at right and moves towards left i.e. in the right to left order sum1 produces a value 48 and sum2 produces 46 as their results.

In case if parenthesized sub-expressions are evaluated first, both sum1 and sum2 results in the values 46.

If functions are evaluated first, both sum1 and sum2 produces 48 as their final value.

SOLVED PROBLEM 2

Assume the following rules of associativity and precedence for expressions.

Table 5.2.3 Associativity and Precedence Rules for Expressions

Precedence	Highest	*,	/,	not			
		+,	-,	&,	mod		
		- (unary)					
		=,	/=,	<,	<=,	>=,	>
		and,					
	Lowest	or,	Xor				
Associativity	Left to right						

Show the order of evaluation of the following expressions,

(i) a + b * c + d **(ii) a * b - 1 + c**

(iii) a * (b - 1)/c mod d **(iv) (a - b)/c & (d * e/a - 3)**

LUTION

(i) **a + b * c + d :** As multiplication operator (*) has higher precedence over addition operator (+), it is evaluated before performing addition operation. Moreover associativity is from left to right, the evaluation proceeds as follows,

- Initially, multiplication operation is performed between the variables 'b' and 'c' (i.e., b * c).
- The product of b and c is added to a (i.e., a + b * c).
- Finally, the result obtained in step-2 is added to d (i.e., a + b * c + d).

(ii) **a * b – 1 + c :** The evaluation is as follows,

- First, the variables a and b are multiplied, as '*' has higher precedence over '+' and '–' (i.e., a * b).
- A value 1 is then subtracted from (a * b) i.e., [(a * b) – 1].
- Finally, c is added to the resulted expression from step-2 (i.e., [(a * b) – 1] + c).

(iii) **a * (b – 1)/c mod d**

- Initially, (b – 1) is performed followed by the multiplication of a and (b – 1) (i.e., a * (b – 1)). This is done as (b – 1) is shown in parentheses.
- The resultant expression (obtained in step-1) is then divided by c i.e., a * (b – 1)/c.
- Finally, a 'mod' operation is applied between a * (b – 1)/c and d as ((a * (b – 1)/c) mod d).

(iv) **((a – b)/c) & ((d * e)/(a – 3))**

- On moving from left to right, first (a-b) is evaluated. The result is then divided by c i.e., (a – b)/c.
- Next, multiplication is applied between d and e (d * e).
- A value 3 is subtracted from a i.e., (a – 3).
- The product of d and e is then divided by (a – 3) i.e., (d * e)/(a – 3).
- Finally, '&' operation is carried out between ((a – b)/c) and ((d * e)/(a – 3)) i.e., [((a – b)/c) & ((d * e)/(a – 3))].

5.3 OVERLOADED OPERATORS

Arithmetic operators are often used for more than one purpose. For e.g., + is frequently used for addition of any numeric type operands. Some languages, Java for example, also use it for string concatenation. This multiple use of an operator is called operator overloading and is generally though to be acceptable, as long as readability and/or reliability do not suffer. Some believe there is too much operator overloading in APL and SNOBOL, where most operators are used for both unary and binary operations.

As an example let us consider the operator ampersand (&) in C. As a binary operator it denotes bitwise logical AND operation. As a unary operator it specifies the address of a variable. For e.g., the execution of,

```
x = &y;
```

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

Virtually all programming languages have a less serious but similar problem, which is often due to the overloading of the minus operator. The problem is only that the compiler cannot tell if the operator is meant to be binary or unary. So once again, failure to include the first operand when the operator is meant to be binary cannot be detected as error by the compiler. However the meanings of the two operations, unary and binary, are atleast closely related, so readability is not affected.

Distinct operator symbols not only increase readability, but they are sometimes convenient to use for common operations as well. The division operator is an example. Consider the problem of finding the floating-point average of a list of integers. Normally the sum of those integers is computed as an integer. Suppose this has been done in the variable sum and the number of values is in count. Now, if the floating-point average is to be computed and placed in the floating-point variable avg, this computation could be specified in C++ as,

```
avg = sum/count;
```

where, avg is floating-point type and sum and count are integer type. Both operands will be implicitly converted to floating point and a floating-point division operation is used. This kind of implicit conversion operation is further discussed in the following section. Integer division in Pascal is specified by the div operator, which takes integer operands and produces an integer result. When no distinct operator for float-point division is provided, explicit conversions must be used.

ome languages that support abstract data types, for e.g., Ada, C++ FORTRAN 90, the programmer to further overload operator symbols. For e.g., suppose a user s to define the * operator between a scalar integer and integer array to mean that element of the array is to multiplied by the scalar. This could be done by writing nction subprogram named * that performs this new operation. The compiler will ose and correct meaning when an over loaded operator is specified, based n the types the operands, as with language-defined overloaded operators. For e.g., if this new finition for * is defined in an Ada program, an Ada compiler will use the new definition r * whenever the * operator appears with a simple integer as the left operand and n integer array as the right operand.

When sensibly used, user-defined operator overloading can aid readability. For e.g., if + and * are overloaded for a matrix abstract data type and A, B, C and D are variables of that type, then,

```
A * B + C * D
```

can be used instead of MatrixAdd(MatrixMult(A, B), MatrixMult(C, D))

On the other hand, user-defined overloading can be harmful to readability. For one thing, nothing prevents a user from defining + to mean multiplication. Furthermore, seeing an * operator in a program, the reader must find both the types of the operands and the definition of the operator to determine its meaning. Any or all of these definitions could be in other files

C++ has a few operators that cannot be overloaded. Among these are the class or structure member operator (.) and the scope resolution operator (::). Operator overloading was one of the C++ features that was not copied into Java.

5.3.1 Solved Problems

SOLVED PROBLEM 1

Explain problems with overloading with suitable example.

SOLUTION

Problems of Operator Overloading

(1) It is easier for the programmer to misunderstand the meaning of an overloaded operator.

(2) Operator overloading affects readability and reliability.

(3) Size of operator, conditional operator cannot be overloaded.

(4) Operator overloading can only change the semantics of an operator but not its syntax.

Example for Operator Overloading

```
//Overloading of unary minus operator.
#include<iostream.h>
class Example
{
    int i, j;
    public:
    void getvalues(int x, int y)
    {
        i = x;
        j = y;
    }
    void show( )
    {
        cout<<"the value of i is"<<i<<endl;
        cout<<"the value of j is"<<j<<endl;
    }
    void operator–( );
};
void example : : operator–( );
{
    i = –i;
    j = –j;
}
int main( )
{
    example e, e1;
    e.getvalues(–5, –6);
    cout<<e.show( );
```

```
e1.getvalues(-8, 2);
e1.show( );
-e;
e.show( );
-e1;
e1.show( );
return 0;
}
```

Output

```
The value of i is - 5
The value of j is - 6
The value of i is - 8
The value of j is 2
The value of i is 5
The value of j is 6
The value of i is 8
The value of j is - 2
```

Problems of Function Overloading

(1) Excessive use of the overloaded functions makes a program unreadable.

(2) Ambiguities between the versions of the overloaded function arises when ver few or too many versions are declared.

(3) All the overloaded versions of a function require separate definitions.

(4) Functions having different scopes cannot be overloaded.

Example for Function Overloading

```
#include<iostream.h>
int multiply(int, int);
double multiply(double, double);
double multiply(double, int);
void main( )
{
```

```
    cout<<"Call to the first function"<<multiply(10, 5);
    cout<<endl<<"Call to the third function"<<multiply(2.0, 4);
    cout<<endl<<"Call to the second function"<<multiply(1.0, 5.0)<<endl;
    cout<<"Call to the first function"<<multiply(2, 4)<<endl;
    cout<<"Call to the second function"<<multiply(5.0, 15.0)<<endl;
    cout<<"Call to the third function"<<multiply(1.0, 1)<<endl;
}
int multiply(int x, int y)
{
    return (x*y);
}
double multiply(double x, double y)
{
    return (x*y);
}
double multiply(double x, int y)
{
    return (x*y);
}
```

Output

```
Call to the first function 50
Call to the third function 8.0
Call to the second function 5.0
Call to the first function 8
Call to the second function 75.0
Call to the third function 1.0
```

SOLVED PROBLEM 2

Consider the following C program,

```
int fun(int * i)
{
    * i + = 5;
    return 4;
}
void main( )
{
    int x = 3;
    x = x + fun(&x);
}
```

What is the value of x after assignment statement in main method assuming,

(i) Operands are evaluated left to right

(ii) Operands are evaluated right to left?

SOLUTION

(i) Operands are Evaluated Left to Right : When the expression x = x + fun(&x) is evaluated from left to right, then the value of x becomes 7.

Explanation : In the left to right evaluation of the expression, x = x + fun(&x), the first operand is evaluated first and second operand is evaluated second i.e. function fun( ) is called after x is assigned a value 3. Here, the function fun( ) is called by pass-by-reference which increments the value of x by 5 and always returns a value 4. However, the change in the value of x does not reflect to the main method since the assignment of x is done before calling the function.

∴ The value of x = x + fun(&x)

= 3 + 4

= 7

(ii) **Operands are Evaluated from Right to Left :** When the expression x = x + fun(&x) is evaluated from right to left, the value of x becomes 12.

Explanation : In the right to left evaluation of the expression, x = x + fun(&x), the second operand is evaluated first and first operand is evaluated second i.e. function 'fun( )' is called before the assignment of x. Again the function fun( ) is called by pass-by-reference which increments the value of x by 5 and returns 4. But, this time the change in the value of x (x = 8) reflects to the main method. Since the assignment of x is done after calling the function.

∴ The value of x = x + fun(&x)

= 8 + 4

= 12

5.4 TYPE CONVERSIONS

Type conversion or Type casting refers to changing an entity of one data type into another. This is done to take certain features of type hierarchies. Type conversions are either narrowing or widening.

Narrowing Conversion : It converts an object to type that cannot include all of the values of the original type.

Example : Converting from float to int will make the initial decimal point values to be lost.

Widening Conversion : It converts an object to a type that can include atleast approximations to all of the values of the original type.

Example : Converting from int to float will not loose any data and as keep approximate precision.

There are two types of conversion, Implicit and Explicit. The term of implicit type conversion is coercion. The most common form of explicit type conversion is known as casting. Explicit type conversion can also be achieved with separately defined conversion routines such as an overloaded object constructor.

Each programming language has its own rules on how types can be converted. In general, both objects and fundamental data types can be converted,

(1) **Implicit Type Conversion :** Implicit Type Conversion, also known as coercion, is an automatic type conversion by the compiler. Some languages allow, compilers to provide coercion.

In a mixed-type expression, data of one or more subtypes can be converted to a supertype as needed at runtime so that the program will run correctly.

Example : The following is legal C language code,

```
double d;
long l;
int i;
if(d > i)        d = i;
if(i > l)        l = i;
if(d == l)       d* = 2;
```

Although d, l and i belong to different data types, they will be automatically converted to equal data types, each time a comparison is executed. This behavior should be used with caution, as unintended consequences can arise. Data can be lost when floating-point representations are converted to integral representations as the fractional components of the floating-point values will be truncated (rounded down). Conversely, converting from an integral representation to a floating-point one can also lose precision, since the floating-point type may be unable to represent the integer exactly. This can lead to situations such as storing the same integer value into two variables of type integer and type real which return false if compared for equality.

(2) Explicit Type Conversion : Explicit type conversion is a type conversion which is explicitly defined within a program (instead of being done by a compiler for implicit type conversion).

Using Casting

```
double da = 5.5;
double db = 5.5;
int result = static_cast<int>(da) + static_cast<int>(db);
        //Result would be equal to 10 instead of 11
```

There are several kinds of explicit conversion.

Checked : Before the conversion is performed, a runtime check is done to see if the destination type can hold the source value. If not, an error condition is raised.

Unchecked : No check is performed. If the destination type cannot hold the source value, the result is undefined.

In explicit type conversion, we proceed the expression with a new type.

Example : I = (int)7.5;

This statement convert it in to an integer i.e., 7.5 is converted in 7.

Operation	ADA		C		FORTRAN 95		Description
	Operator	Example	Operator	Example	Operator	Example	
Equal	=	x = y	==	x == y	.EQ. or ==	x.EQ.y or x == y	x is equal to y
Not equal	/=	x /= y	!=	x != y	.NE. or <>	x.NE.y or x <> y	x is not equal to y
Greater than	>	x > y	>	x > y	.GT. or >	x.GT.y or x > y	x is greater than y
Less than	<	x < y	<	x < y	.LT. or <	x.LT.y or x < y	x is Less than y
Greater than or equal	>=	x >= y	>=	x >= y	.GE. or >=	x.GE.y or x >= y	x is greater than or equal to y
Less than or equal	<=	x <= y	<=	x <= y	.LE. or <=	x.LE.y or x <= y	x is Less than or equal to y

he relational operators have lower precedence over arithmetic operators.

5.2 Boolean Expressions

An expression consisting of boolean variables, boolean constants, relation pression and boolean operators is called a 'boolean expression'. Usually, the operato ich as AND, OR and NOT are used in boolean expressions. Boolean expression alwa eturn boolean values.

The AND and OR operators have equal precedence in boolean algebra. Hence, the have equal precedence in Ada whereas, in C-based languages, the AND operator ha higher precedence over OR operator. Because, the arithmetic expressions can be use as operands in relational expression which in turn can be used as operands in boolea expressions, they have different precedence levels.

The highest to lowest precedence of arithmetic, relational and boolean operator in C-based languages is given below,

High Precedence
Postfix ++, – –
Unary +, –, prefix ++, – –, !
*, /, %
Binary +, –
<, >, <=, >=
=, /=
&&
Low Precedence
||

Earlier versions of C (prior to C99) do not have any boolean data type hence, doesn't support any boolean type value. Instead numeric variables and constants representing 0 as false and nonzero values as true are considered. The result of such an evaluation is also in terms of zeros and non-zeros representing false and true respectively.

5.6 SHORT-CIRCUIT EVALUATION

It is a method in which the value of an expression can be deduced without evaluating all the operands and the operators. A compiler that performs short-circuit evaluation of boolean expressions will result in a code that skips the second comparison in an expression when the overall value can be computed from the first half itself.

Example

(1) Consider the boolean expression (x < y) and (y < z),

If x is greater than y, there is really no point in checking to see whether y is less than z because we know that the overall expression is false.

Similarly, for an expression (x > y) or (y > z),

If x is greater than y, there is no need to check whether y is greater than z, as we know that the overall expression must be true.

(2) Consider the arithmetic expression (2 * a) * (b/2 – 1),

The above expression is independent of the value of (b/2 – 1) if a is 0, i.e., when a is 0, there is no need to calculate (b/2 – 1) or to perform the second multiplication. This shortcut in arithmetic expression is not detected easily during execution so, it results in side effects. If program correctness depends on the side effects, short-circuit evaluation can result in a serious error.

Advantages

(1) It is the best design as it provides flexibility to the programmer in choosing short-circuit evaluation for some or all the boolean expressions.

Example : Consider the boolean expression (x < y) and (y < z).

If x is greater then y, here is really no point in checking to see whether y is less than z since it implies that the overall expression is false.

(2) It helps to determine the result of the expression without evaluating all the operands and/or operators.

Example : For arithmetic expression (a * b)*(c/4), the result is independent of the value (c/4) if a = 0. Thus, there is no need to evaluate (c/4).

(3) It is possible to specify the condition for testing the validity of an array index within the expression defined for testing the subscripted value:

Example : If (i < =. 1st_index and arr[i] > =. a)

In the above example, the condition i < = 1st_index will prevent the occurrence of an error if the value of last index is less than i.

(4) It possible to specify the condition for testing a null pointer in the expression defined for dereferencing that pointer.

Example : If (a !=null and b% a==null)

The above example test whether the value of variable 'a' is equal to null or not. This type of evaluation doesn't generate run-time error since the condition al = null is executed prior to the second condition b% a==null.

5.7 ASSIGNMENT STATEMENTS

An assignment statement is a way of changing the values of the variables, dynamically. The assignment statement is one of the central constructs in imperative languages.

Various alternatives of the assignment statement are described as follows,

5.7.1 Simple Assignments

The simple assignment statements have the following general form,

<destination_variable><assignment_operator><expression>

Example : a = x + y;

An equal sign is used to represent the assignment operator in the languages like Fortran, Basic and C-based languages.

The equality relational operator is written as '= =' in C-based languages to avoid the confusion with the assignment operator.

5.7.2 Conditional Targets

The C-based languages, such as C++, Java and C# uses conditional targets on the assignment statements as follows,

```
a ? b : c = 0;
```

which is equivalent to,

```
if(a)
    b = 0;
else
    c = 0;
```

5.7.3 Compound Assignment Operators

An assignment statement in which the destination variable also appears on the right hand side, as its first operand, is called a compound assignment statement. This can be done by using a compound assignment operator.

The use of a compound assignment operator provides a shorthand representation for the most commonly needed assignment statements.

Example : Consider the following compound assignment statement,

```
x = x + y;
```

ALGOL 68 was the first language that uses the compound assignment operators. Later, they are used by C-based languages in a slightly different form in which an assignment operator is associated with the desired binary operator as in,

```
x + = y;
```

which is equivalent to,

```
x = x + y;
```

5.7.4 Unary Assignment Operators

C, C++ and Java 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 as either prefix operators, meaning they precede the operands, or as postfix operators, meaning they follow the operands. In the assignment statement,

```
x = ++ a;
```

The value of a is incremented by 1 and then assigned to x. This could also be stated as,

```
a = a + 1;
x = a;
```

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

```
x = a ++;
```

The assignment of the value of a to x occurs first, then a is incremented. The effect is the same as that of the two statements,

```
x = a;
a = a + 1;
```

An example of the use of the unary increment operator to form a complete assignment statement is,

```
a ++;
```

Which simply increments a. It does not look like an assignment, but it certainly is one. It is equivalent to the statement,

```
a = a + 1;
```

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

Example

```
- a ++ ;
```

a is first incremented and then negated. So it is,

```
- (a ++) ;
```

not

```
(-a) ++ ;
```

The increment and decrement operators of C, C++ and Java are frequently used to form array

5.7.5 Assignment as an Expression

An assignment statement in C-based languages generates a result that is equivalent to the value assigned to the target variable. Hence, it can either be used as an expression or as an operand in some other expressions. This type of assignment is similar to the binary operator and produces a side effect of changing its left operand.

Example :
```
while((C = getchar()) != EOF)
{
    statements;
}
```

In the above statement, the next character is obtained from the keyboard by using the getchar( ) function. The accepted character is then assigned to variable C. The resultant value is then compared with EOF. In case, if C is not equal to EOF, the compound statement associated with the while is executed. As the assignment operator has lower precedence than the relational operators in these languages it must be parenthesized. If the parentheses are absent then, the newly entered character is first compared with EOF, and the resultant boolean value (either 0 or 1) is assigned to the variable C.

When assignments statements are used as the operands in expressions, they produce a side effect in which they affects the readability and understandability of the expressions.

C uses an assignment operator which causes the assignments of multiple targeted statements as follows,

```
a = b = 0;
```

In which b is initially assigned to 0 and then its value is assigned to a.

Loss of error detection occurs when an assignment operator (=) is used in place of comparison operator (==). This is a mistake but not an error to be detected by a compiler because instead of testing a relational expression, the value assigned to the variable is tested.

5.7.6 List Assignments

Perl and Ruby programming languages provide multiple target multiple-source assignment statements.

Example : Consider a statement in Perl,

```
($Maths, $science) = (40, 20);
```

The above statement states that 40 is assigned to $maths, and 20 is assigned to $science. If the values of two variables must be interchanged, this can be done with a single statement as.

```
($Maths, $Science) = ($Science, $Maths);
```

In Perl, if there are more values on the left side than variables on right side, they are set to under. If there are more values on the right side than variables on left side. The excess values are ignored. The syntax in Ruby is similar to that of Perl except the left and right sides are not parenthesized.

5.8 MIXED-MODE ASSIGNMENT

Mixed-mode expressions are those expressions in which an operator takes the operands of different types, they must define coercions which are the conventions for implicit operand type conversions because the binary operations in computers does not take operands of different types.

Assignment statements are also mixed-mode. In mixed-mode assignment statements, it is important to determine whether the type of expression and the type of variable being assigned, matches or not.

The coercion rules are used by Fortran and the C-based languages for providing mixed-mode assignments. However, Ada doesn't allows mixed-mode assignment. Mixed-mode assignment is allowed in Java and C# only if the coercion is widening. Hence, a value of int type can be assigned to a variable of type float but the opposite is not true.

The process of coercion takes place only after evaluating the right side expression, n all the languages that supports mixed-mode assignments. This can be done by coercing all the operands on the right side, to the target type before evaluating the expression.

Example

```
int x, y;
float z;
:
z = x/y;
```

The values of x and y are coerced to float before performing division operation. This s done because z is a float type variable. If the process of coercion delays then a different value for z is produced.

ntages

Usage of mixed-mode expressions reduces the occurrence of errors. Hence, A
provides only a few mixed type operands to be used in expressions.

Other programming languages do not impose any limitations (restrictions) on mixe
mode arithmetic type of expressions.

Mixing of many arithmetic expressions are allowed in Fortran and C-based language
by applying the coercion rules.

vantages

ada doesn't allow integer and floating point operands mixing in any expression usin
single exception.

nly few types of mixing related to subrange types is allowed in ADA.

❖ ❖ ❖