UNIT - II

Data Types

Unit-02/Lecture-01

A data type defines a collection of data values and a set of predefined operations on those values. Types provide implicit context. Compilers can infer information, so programmers write less code.

e.g., the expression a+b in Java may be adding two integer, two floats or two strings depending on context.

Types provide a set of semantically valid operation. Compilers can detect semantic mistakes

e.g., Python's list support append () and pop (), but complex numbers do not.

Design Issues for all Data Types

- How is the domain of values specified?
- What operations are defined and how are they specified?
- What is the syntax of references to variables?

Typical primitives include: Boolean, Character, Integral Type, Fixed point type, Floating point type.

Primitive Data Types

• Almost all programming languages provide a set of primitive data types

•Primitive data types: Those not defined in terms of other data types

•Some primitive data types are merely reflections of the hardware

•Others require little non-hardware support

Primitive Data Types: Integer

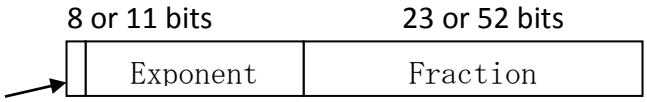
- The most common primitive numeric data type is integer.
- Many computers support several sizes of integers, and these capabilities are reflected in some programming languages For example, Ada allows these: short integer, integer and long integer.

- An integer is represented by a string of bits, with the leftmost representing the sign bit.

FLOATING-POINT

- Floating point data types model real numbers, but the representations are only approximations for most real values.
- On most computers, floating-point numbers are stored in binary, which exacerbates the problem
- Floating-point values are represented as fractions and exponents
- Most new computers use the standard IEEE format
- Most languages use float and double as floating-point types
- The float is stored in 4 bytes of memory
- The double has twice as big of storage.

IEEE Floating Point Standard 754



Sign bit

Precision - The accuracy of the fractional part of a value, measured as the number of bits

Range – a combination of the range of fractions and the range of exponents.

Primitive Data Types: Complex

•Represented as an ordered pair of floating numbers

•Python specifies the imaginary part by following it with a j or J(7 + 3j)

•Languages that support a complex type include operations for arithmetic on complex values

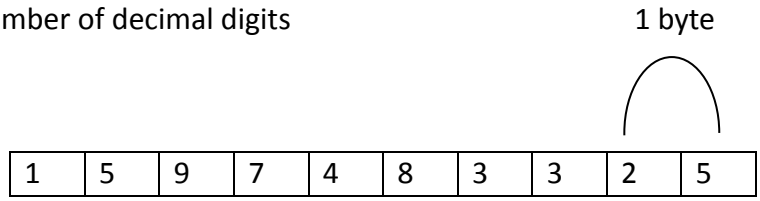
Primitive Data Types: Decimal

• For business applications (money)

–Essential to COBOL

–C# offers a decimal data type

•Store a fixed number of decimal digits



BCD code

•Advantage: accuracy

•Disadvantages: limited range, wastes memory

Primitive Data Types: Boolean

•Simplest of all

•Range of values: two elements, one for “true” and one for “false”

•Could be implemented as bits, but often as bytes.

•Advantage: readability (compared with using integers to represent switches/flags)

Primitive Data Types: Character

•Stored as numeric coding

•Most commonly used coding: ASCII

• An alternative, 16-bit coding: Unicode

–Includes characters from most natural languages

–Originally used in Java

–C# and JavaScript also support Unicode

Unit-02/Lecture-02

CHARACTER STRING TYPES

- A character string type is one in which the values consist of sequences of characters.
- They are used to label output, and input and output of all kinds.
- Design Issues-
  - Is it a primitive type or just a special kind of array?
  - Is the length of objects fixed or variable?
- Operations-
  - Assignment
  - Comparison (=, > etc)
  - Concatenation
  - Substring reference
  - Pattern matching

Array Types

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

Array Design Issues

- What types are legal for subscripts?
- Are subscripting expressions in element references range checked?
- When are subscript ranges bound?
- When does allocation take place?
- What is the maximum number of subscripts?
- Can array objects be initialized?
- Are any kinds of slices allowed?

Array Indexing

•Indexing (or subscripting) is a mapping from indices to elements

array_name (index_value_list) $\rightarrow$element

•Index Syntax

–FORTRAN, PL/I, Ada use parentheses

- Ada explicitly uses parentheses to show uniformity between array references and

function calls because both are mappings

–Most other languages use brackets

Associative Arrays

An associative array is an array with strings as index. This stores element values in association with key values rather than in a strict linear index order. In associative arrays the keys are not integers but strings.

The associative arrays are very similar to numeric arrays in term of functionality but they are different in terms of their index. Associative array will have their index as string so that you can establish a strong association between key and values.

To store the salaries of employees in an array, a numerically indexed array would not be the best choice. Instead, we could use the employees names as the keys in our associative array, and the value would be their respective salary.

```
$aFruit = array(
        'color' => 'red'
      , 'taste' => 'sweet'
      , 'shape' => 'round'
      , 'name'  => 'apple'
      ,       4     // key will be 0
      );
```

Is equivalent with:

```
$aFruit['color'] = 'red';
$aFruit['taste'] = 'sweet';
$aFruit['shape'] = 'round';
$aFruit['name'] = 'apple';
$aFruit[]       = 4;       // key will be 0
```

User Defined Ordinal Types

An ordinal type is one in which the range of possible values can be easily associated with the set of positive integers .Two common kinds:

- Enumeration types
- Sub range types

Enumeration Types

The user enumerates all of the possible values, which are symbolic constants

Design Issue: Should a symbolic constant be allowed to be in more than one type definition?

Examples:

Pascal--cannot reuse constants; they can be used for array subscripts, for variables, case selectors; no input or output; can be compared

Ada--constants can be reused (overloaded literals); can be used as in Pascal; input and output supported

C and C++--like Pascal, except they can be input and output as integers

Java does not include an enumeration type

In other words, an enumeration is a list of values.

```
type TWeekDays = (Monday, Tuesday, Wednesday,
                  Thursday, Friday, Saturday, Sunday);
```

Once we define an enumerated data type, we can declare variables to be of that type:

```
var SomeDay : TWeekDays;
```

Sub range Types

- An ordered, contiguous subsequence of another ordinal type
- Design Issue: How can they be used?

Examples:

- Pascal--sub range types behave as their parent types; can be used as for

variables and array indices

type pos = 0 .. MAXINT;

- Ada--subtypes are not new types, just constrained existing types (so they are compatible); can be used as in Pascal, plus case constants

subtype Pos_Type is

Integer range 0...Integer ‘Last;

Unit-02/Lecture-03

Record

A record is a data structure composed of a fixed number of components of different types. The components may be heterogeneous, and they are named with symbolic names.

Specification of attributes of a record:

Number of components
Data type of each component
Selector used to name each component.

Implementation:

Storage: single sequential block of memory where the components are stored sequentially.

Selection: provided the type of each component is known, the location can be computed at translation time.

Note on efficiency of storage representation:

For some data types storage must begin on specific memory boundaries (required by the hardware organization). For example, integers must be allocated at word boundaries (e.g. addresses that are multiples of 4). When the structure of a record is designed, this fact has to be taken into consideration. Otherwise the actual memory needed might be more than the sum of the length of each component in the record. Here is an example:

```
struct employee

  { char Division;

    int  IdNumber; };
```

The first variable occupies one byte only. The next three bytes will remain unused and then the second variable will be allocated to a word boundary. Careless design may result in doubling the memory requirements.

Unions

A union is an aggregate variable that can declare several data types, but only store the value of one variable at any one time; each data type shares the same area of memory. The declaration of a union is similar to that of a structure:

```
union [union_tag]
{
    data_type variable_1;
    data_type variable_2;
    data_type variable_3;
    .
    .
    .
} {union_variable_name};
```

As with a structure, a union_tag is optional if a union_variable_name is present. The compiler allocates only the storage space of the largest data_type declared.

```
union Data
{
    char str[6];
    int y;
    long z;
    float x;
    double t;
} var;
```

The union variable var is given eight(8) bytes of storage; all the elements start at the same address in memory; the compiler gives the amount of storage needed by the largest data

type, in this case double which is eight bytes.

Accessing Members of a union

To access a member of a union the dot operator is used the same as with structures.

```
var.x = 123.45;
```

Unions passed to functions exactly as a structure would be passed.

Pointers and Reference Types

Pointers and references are essentially variables that hold memory addresses as their values. Pointers and references hold the addresses in memory of where we find the data of the various data types that we have declared and assigned. The two mechanisms, pointers and references, have different syntax and different traditional uses.

A pointer is declared as:

<Pointer type> *<pointer-name>

In the above declaration:

1. Pointer-type: It specifies the type of pointer. It can be int, char, float etc. This type specifies the type of variable whose address this pointer can store.
2. Pointer-name: It can be any name specified by the user. Professionally, there are some coding styles which every code follows. The pointer names commonly start with ‘p’ or end with ‘ptr’

```
int x = 1, y = 2;

int *ip;

ip = &x;
```

A reference is treated *exactly* as if we had used the original variable in its place. For example, if we assign to a reference, it is as if we assigned to the original variable:

```
int x = 5;

int &y = x;  // y is an alias for x

y = 6;  // now x == 6
```

Unit-02/Lecture-04

Names

Names are also associated with labels, subprograms, formal parameters and other program constructs.

Name Forms-

A name is a string of characters used to identify some entity in a program. The earliest programming languages used single character names.

FORTRAN1 broke this tradition by allowing names up to 6 characters long. FORTRAN90 and C allow up to 31 characters names. ADA has no length limit.

We can use some connecter characters like underscore (_) in the string. Some languages like C, C++, a Java are case sensitive. That is, these languages differentiate between uppercase and lowercase letters. Ex- SUN, Sun and sun are distinct in C++.

Special words- Special words are used in programming languages to make programs more readable by naming actions to be performed.

Keyword-

A keyword is a word of programming languages that is special only in certain contexts. FORTRAN is one of the languages whose special word are keywords. For Ex- In FORTRAN, the word real when found in the beginning of a statement and followed by a name is considered a keyword that indicates the statement is a declarative statement. However if the word real is followed by an assignment operator it considered as a variable name.

Ex- REAL APPLE

REAL = 3.4

Reserved Word- A reserved word is a special word of a programming language that cannot be used as name.

Ex- The names printf and scanf are the reserved words which are defined in <stdio.h>

VARIABLES

A variable is a symbolic name for (or reference to) information. The variable's name

represents what information the variable contains. They are called variables because the represented information can change but the operations on the variable remain the same. In general, a program should be written with "Symbolic" notation, such that a statement is always true symbolically.

For example, if we want to find the sum of any two numbers we can write:

result = a + b;

Both 'a' and 'b' are variables. They are symbolic representations of any numbers. For example, the variable 'a' could contain the number 5 and the variable 'b' could contain the number 10. During execution of the program, the statement "a + b" is replaced by the Actual Values "5 + 10" and the result becomes 15.

Variable Properties

There are 6 properties associated with a variable.

1. A Type
2. A Value
3. A Scope
4. A Life Time
5. A Location (in Memory)

Properties

1. A Name

The name is Symbolic. It represents the "title" of the information that is being stored with the variable.

The name is perhaps the most important property to the programmer, because this is how we "access" the variable. Every variable must have a unique name!

2. A Type

The type represents what "kind" of data is stored with the variable.

In C, Java etc, the type of a variable must be explicitly declared when the name is created.

3. A Value

A variable, by its very name, changes over time. Thus if the variable is jims_age and is assigned the value 21. At another point, jims_age may be assigned the value 27.

Default Values

Most of the time, when we "create a variable" w are primarily defining the variables name and type. Often we will want to provide an initial value to be associated with that variable name. If you forget to assign an initial value, then various rules "kick in" depending on the language.

Example

```
age = 20; //this creates a variable named age with the value 20 (and type Numbe
r (double))

fprintf('age is %f\n');    % answer: age is 20
```

4. A Scope

Good programs are "Chopped" into small self contained sections (called functions). A variable that is seen and used in one function is NOT available in another section. This allows us to reuse variable names, such as age. In one function 'age' could refer to the age of a student, and in another function 'age' could refer to the vintage of a fine wine.

Further this prevents us from "accidentally" changing information that is important to another part of our program.

5. A Life Time

The life time of a variable is strongly related to the scope of the variable. When a program begins, variables "come to life" when the program reaches the line of code where they are "declared". Variables "die" when the program leaves the "Scope" of the variable.

6. A Location (in Memory)

Generally we don't have to worry too much about where in the computer hardware the variable is stored. The computer does this for us. But we should be aware that a "Bucket" or "Envelope" exists in the hardware for every variable we declare. In the case of an array, a "bunch of buckets" exist. Every bucket can contain a single value.

Unit-02/Lecture-05

Concept of Binding

A binding is an association, such as between an attribute and an entity, or between an operation and a symbol. Binding time is the time at which a binding takes place

For example, in C the binding time for a variable type is when the program is compiled (because the type cannot be changed without changing and recompiling the program), but the value of the variable is not bound until the program executes (that is, the value of the variable can change during execution).

Some additional examples of attributes are:

- the meaning of a keyword such as if
- the operation associated with a symbol such as +
- the entity (variable, keyword, etc.) represented by an identifier
- the memory location for the value of an identifier

The most common binding times for attributes are (in chronological order):

1. Language definition
2. Language implementation
3. Program translation (compile time)
4. Link edit
5. Load
6. Program execution (run time)

Classes of Binding Times

1. Execution Time (Run time)

This includes binding performed during program execution. Ex- Binding of variables to their values as well as bindings of variables to particular storage locations.

2. Translation Time (Compile Time)-

a) Binding chosen by the programmer

While writing a program, a programmer gives choice for variable names, types for variables, program statement structures and so on that represents binding during translation.

b) Binding chosen by the translator

Some bindings are chosen by translator. Ex- The relative location of a data object in the storage allocated for a procedure, how arrays are stored, how descriptors for the arrays, if any are created all such decisions are made by the language translator.

c) Binding chosen by the Loader- A program usually consists of several subprograms that must be merged into a single executable program. The translator binds variables to addresses within the storage designated for each subprogram.

3. Language Implementation Time- Some aspects of a language definition may vary between implementations. For Ex- The details associated with the representation of numbers and arithmetic operation may be determined by the underlying computer hardware.
4. Language Definition Time- Most of the structure of programming language is fixed at language definition time. Ex-Different data types, data structure types, control elements, program structure and so on are all fixed at language definition time.

Types of Binding

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

Static Type Binding

• If static, the type may be specified by either an explicit or an implicit declaration.

Variable Declarations

•An explicit declaration is a program statement used for declaring the types of variables.

•An implicit declaration is a default mechanism for specifying types of variables (the first appearance of the variable in the program.)

• Both explicit and implicit declarations create static bindings to types.

•FORTRAN, PL/I, BASIC, and Perl provide implicit declarations.

EX: –In FORTRAN, an identifier that appears in a program that is not explicitly declared is implicitly declared according to the following convention: I, J, K, L, M, or N or their lowercase versions is implicitly declared to be Integer type; otherwise, it is implicitly declared as Real type.

–Advantage: writ ability

–Disadvantage: reliability suffers because they prevent the compilation process from detecting some typographical and programming errors.

–In FORTRAN, variables that are accidentally left undeclared are given default types and unexpected attributes, which could cause subtle errors that, are difficult to diagnose.

Dynamic Type Binding (JavaScript and PHP)

•Specified through an assignment statement

•Ex, JavaScript

list = [2, 4.33, 6, 8]; ->single-dimensioned array

list = 47; -> scalar variable

–Advantage: flexibility (generic program units)

–Disadvantages:

–High cost (dynamic type checking and interpretation)

•Dynamic type bindings must be implemented using pure interpreter not compilers.

•Pure interpretation typically takes at least ten times as long as to execute equivalent machine code.

–Type error detection by the compiler is difficult because any variable can be assigned a value of any type.

•Incorrect types of right sides of assignments are not detected as errors; rather, the type of the left side is simply changed to the incorrect type.

•Ex: i, x -> Integer

y ->floating-point array

i = x ->what the user meant to type

i = y -> what the user typed instead

•No error is detected by the compiler or run-time system. i is simply changed to a floating-point array type. Hence, the result is erroneous. In a static type binding language, the compiler would detect the error and the program would not get to execution.

Unit-02/Lecture-06

TYPE CHECKING

Type systems are the biggest point of variation across programming languages. Even languages that look similar are often greatly different when it comes to their type systems.

Definition: A type system is a set of types and type constructors (integers, arrays, classes, etc.) along with the rules that govern whether or not a program is legal with respect to types (i.e., type checking). For example, C++ and Java have similar syntax and the control structures. They even have a similar set of types (classes, arrays, etc.). However, they differ greatly with respect to the rules that determine whether or not a program is legal with respect to types.

As an example, one can do this in C++ but not in Java:

int x = (int) “Hello”;

In other words, Java’s type rules do not allow the above statement but C++’s type rules do allow it. Why do different languages use different type systems? The reason for this is that there is no one perfect type system. Each type system has its strengths and weaknesses. Thus, different languages use different type systems because they have different priorities. A language designed for writing operating systems is not appropriate for programming the web; thus they will use different type systems. When designing a type system for a language, the language designer needs to balance the tradeoffs between execution efficiency, expressiveness, safety, simplicity, etc. In other words, the type system affects many of the characteristics. Thus, a good understanding of type systems is crucial for understanding how to best exploit programming languages.

Type Conversion and Coercion

Coercion-

Implicit type conversion, also known as coercion, is an automatic type conversion by the compiler. Some languages allow, or even require compilers to provide coercion.

In a mixed type expression, a subtype *s* will be converted to a super type *t* or some subtypes s_1, s_2, ... will be converted to a super type *t* (maybe none of the s_i is of type *t*) at runtime so that the program will run correctly. For example:

double d;

```
long    l;

int     i;

if (d > i)      d = i;

if (i > l)      l = i;

if (d == l)     d *= 2;
```

is legal in a C language program. Although *d*, *l* and *i* belong to different datatypes, they will be automatically converted to the same data type each time a comparison or assignment is executed.

Type Casting

Typecasting is making a variable of one type, such as an int, act like another type, a char, for one single operation. To typecast something, simply put the type of variable you want the actual variable to act as inside parentheses in front of the actual variable. (char)a will make 'a' function as a char.

Used when the programmer wants to explicitly convert one data type to another. It shows the programmer's intention as being very clear.

```
i = 5.2 / f; // warning "loss of precision"

i = int(5.2 / f) // no warning but still loss

f = float(3 * i)
```

TYPE COMPATIBILITY

Compatible Types:

A compatible type is one that is either legal for the operator, or is allowed under language rules to be implicitly converted, by compiler-generated code, to a legal type.

Type compatibility is also called conformance or equivalence.

There are two type compatibility methods:

♣Name compatibility (also called strict compatibility):

Two variables can have compatible types only if they are in either the same declaration or in

declarations that use the same type name.

It is highly restrictive.

It is easier to implement.

Ada uses name compatibility.

♣Structure type compatibility

Two variables have compatible types if their types have identical structures.

It is more flexible.

It is difficult to implement: compare the whole structure instead of just names.

Two types are structurally compatible if:

1. T1 is name compatible with T2; or

2. T1 and T2 are defined by applying the same type constructor to structurally compatible corresponding type components.

Examples:

1. Two records or structure types compatible if they have same structure but different field names?

2. Two single-dimensioned array types in a Pascal or Ada program are compatible if they have the same element type but have different subscript ranges?

3. C uses structural compatibility except for structures.

Unit-02/Lecture-07

Named Constants

A named constant is a variable that is bound to a value only at the time it is bound to a value only at the time it is bound to storage; its value cannot be changed by assignment or by an input statement.

Const int Max =30;

Here Max is a constant of type integer with value30.

Advantages:-

- It improves program readability and reliability. For Ex- using the name 'Pi' in the program is more readable than using the value 3.14.
- Another advantage of named constant is in the program that process a fixed number of data values say 10. Such programs usually use the constant 10 in number of statements like for declaring array subscript ranges, for loop control limits and other uses.

Ex-

```
void main()

{

const int limit=10;

int A[limit], B[limit], C[limit];

cout<<"enter array A elements";

for (i=0; i<limit; i++)

{                       cin>>A[i];

}

cout<<"enter array B elements";

for (i=0; i<limit; i++)
```

```
{                       cin>>B[i];

}

for (i=0; i<limit; i++)

{                       C[i]= A[i]+B[i];

}

cout<<"the addition of array A and B elements";

for (i=0;i<limit;i++)

{                       cout<<C[i];

}

}
```

The advantage of using named constant 'limit' is that when the array limit needs to be changed say from 10 to 100, then only one line is required to be changed, regardless the number of times it is used in the program.

Variable Initialization

The binding of a variable to a value at the time it is bound to storage is called variable initialization.

If the variable is bound to storage statically binding and initialization occur before runtime. If storage binding is dynamic, initialization is also dynamic.

Ex- int fact =1;

Here the variable 'fact' is initialized with the value 1 statically.

Conditional Statements

A conditional statement is one which makes the computer compare two or more variables in some way and decide that the outcome is either 'true' or 'false', and then feeds this into a function such as 'if' or 'while'.

The If statement

If the condition expression evaluates to true, the statement is ignored.

Syntax if(condition)

Statement;

Ex; - if (A < B)

{ printf (“A is smaller element”); }

If..Else Statement

It is used to test the condition that has true and false part.

Syntax: - if (condition)

Statement1;

else

Statement2;

Ex- if (A < B)

{ printf (“A is smaller element”);

}

else

{

printf “B is smaller element”);

}

Switch Statement

A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each switch case.

Syntax:

The syntax for a switch statement in C programming language is as follows:

```
switch(expression){
   case constant-expression  :
      statement(s);
      break; /* optional */
   case constant-expression  :
      statement(s);
      break; /* optional */

   /* you can have any number of case statements */
   default : /* Optional */
      statement(s);
}
```

The?: operator

It is shorthand method for specifying

(If expression)? (Evaluate if true): (else evaluate this)

This reduces the readability of program. This does not in any way speed up execution time.

Unit-02/Lecture-08

LOOPS

Loop statements are used to perform some action repeatedly. Different types of loop control statement used in C language are-

1) While statement – The while statement is used to carry out looping operations in which a group of statements is executed repeatedly until some condition has been satisfied.

while (expression)

{

Statement

}

Ex- To print numbers between 1 to 100 using while

```
#include<stdio.h>
void main()
{
                int i=1;
                while (i<=100)
                {
                    printf("\n %d",i);
                    i++;
}

}
```

2) The do while statement- When a loop is constructed using the while statement the test for continuation of the loop is carried out at the beginning of each pass. Sometime it is desirable to have a loop with the test for continuation at the end of each pass.

do

{

Statement

} while (expression);

The statement will be executed repeatedly, till the value of the expression is true.

Ex-

```
 void main()
{
Int i=1;
do
{
                        printf( “%d”, i);
                        I++;
} while (i<=100);
```

3) The for statement

The for statement is the most commonly used looping statement in C

```
for (expression1; expression2; expression3)

                        Statement;
```

where expression1 is used to initialize some parameter that controls the looping action. Expression2 represents a condition that must be true to continue execution and expression3 is used to alter the value of the parameter.

Ex-

```
 void main()

{

Int i=1;

      For ( i=1;i<=100; i++)

      {

                        Printf(“%d”, i);

       }
```

Unit-02/Lecture-09

SEQUENCE CONTROL

Control Structure in a PL provides the basic framework within which operations and data are combined into a program and sets of programs.

Sequence Control-> Control of the order of execution of the operations

Data Control-> Control of transmission of data among subprograms of program

Sequence Control may be categorized into four groups:

1) Expressions–

They form the building blocks for statements. An expression is a combination of variable constants and operators according to syntax of language. Properties as precedence rules and parentheses determine how expressions are evaluated

2) Statements–

The statements (conditional & iterative) determine how control flows from one part of program to another.

3) Declarative Programming–

This is an execution model of program which is independent of the program statements. Logic programming model of PROLOG.

4) Subprograms–In structured programming, program is divided into small sections and each section is called subprogram. Subprogram calls and co-routines, can be invoked repeatedly and transfer control from one part of program to another.

IMPLICIT AND EXPLICIT SEQUENCE CONTROL

Implicit Sequence Control

Implicit or default sequence control structures are those defined by the programming language itself. These structures can be modified explicitly by the programmer.

eg. Most languages define physical sequence as the sequence in which statements are executed.

Explicit Sequence Control

Explicit sequence control structures are those that programmer may optionally use to modify the implicit sequence of operations defined by the language.

eg. Use parentheses within expressions, or goto statements and labels

Sequence Control within Expressions

Expression is a formula which uses operators and operands to give the output value.

i) Arithmetic Expression –An expression consisting of numerical values (any number, variable or function call) together with some arithmetic operator is called “Arithmetic Expression”.

Evaluation of Arithmetic Expression

Arithmetic Expressions are evaluated from left to right and using the rules of precedence of operators. If expression involves parentheses, the expression inside parentheses is evaluated first.

ii) Relational Expressions –An expression involving a relational operator is known as “Relational Expression”. A relational expression can be defined as a meaningful combination of operands and relational operators.

(a + b) > c c < b

Evaluation of Relational Expression

The relational operators <, >, <=, >= are given the first priority and other operators (== and ! =) are given the second priority. The arithmetic operators have higher priority over relational operators. The resulting expression will be of integer type, true = 1, false = 0

iii) Logical Expression –

An expression involving logical operators is called ‘Logical expression”. The expression formed with two or more relational expression is called logical expression.

Ex. a > b && b < c

Evaluation of Logical Expression

The result of a logical expression is either true or false. For expression involving AND (&&), OR (||) and NOT (!) operations, expression involving NOT is evaluated first, then the expression with AND and finally the expression having OR is evaluated.

Sequence Control within Expressions

1 Controlling the evaluation of expressions

a) Precedence (Priority)

If expression involving more than one operator is evaluated, the operator at higher level of precedence is evaluated first

b) Associativity

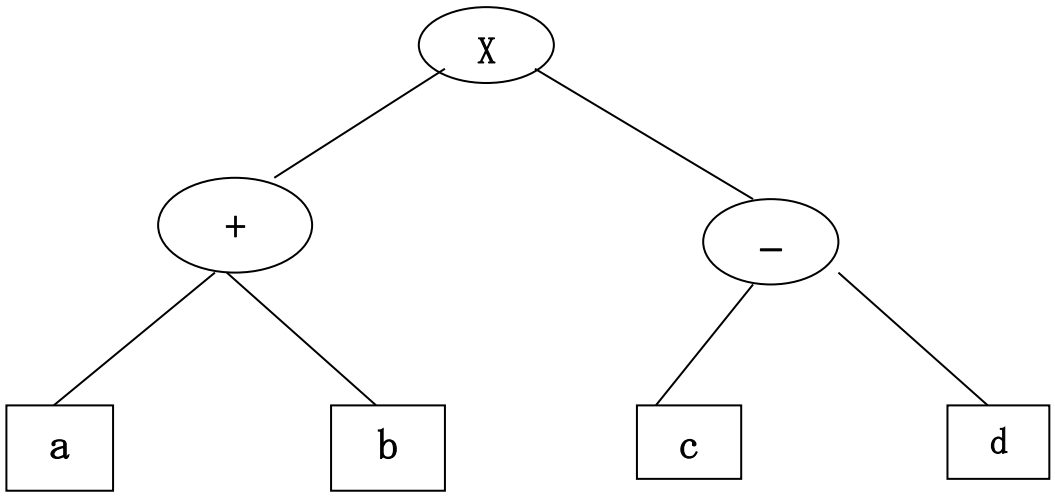
The operators of the same precedence are evaluated either from left to right or from right to left depending on the level. Most operators are evaluated from left to right except

+ (unary plus), -(unary minus) ++, --, !, &

Assignment operators = , +=, *=, /=, %=

Expression Tree

An expression (Arithmetic, relational or logical) can be represented in the form of an "expression tree". The last or main operator comes on the top (root). Example: (a + b) * (c – d) can be represented as



Syntax for Expressions

a) Prefix or Polish notation

Named after polish mathematician Jan Lukasiewicz, refers to notation in which operator symbol is placed before its operands.

*XY, -AB, /*ab -cd

Cambridge Polish- variant of notation used in LISP, parentheses surround an operator and its arguments.

(/(*ab)(-cd))

b) Postfix or reverse polish

Postfix refers to notation in which the operator symbol is placed after its two operands.

AB*, XY-

c) Infix notation

It is most suitable for binary (dyadic) operation. The operator symbol is placed between the two operands.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Why pointer is necessary in any programming language?	JUNE 2011	5 marks
Q.2	Define pointer. Explain various design issues of pointer and pointers in C/ C++.	JUNE 2014	7 marks
Q.3	What is Sequence Control? Explain various categories of sequence control	JUNE 2014	7 marks
Q.4	Explain the different categories of scalar type variables with their advantages and disadvantages.	JUNE 2014	7 marks
Q.5	Explain how dynamic type checking affects system performance and improves flexibility with example.	JUNE 2013	10 marks
Q.6	Differentiate records from variant records with suitable example?	JUNE 2013	10 marks
Q.7	Explain the following terms: Strong typing Type Coercion Pointers	JUNE 2013	10 marks