# Type checking & Type Conversion

Type checking means checking that each operation should receive proper no. of arguments and of proper data type. like

$$A = B * J + d;$$

* and - are basically int and float datatype based operations and if any variable in this A = B*J+d; is of other than int and float then compiler will generate **type error**.

## Two Ways of Type checking :-

1) **Dynamic Type checking :-** It is done at runtime.

→ It uses concept of typetag which is stored in each data objects that indicates the data type of the object.

**Eg:-** an integer data object contains its 'type' and 'value' attribute.

So operation only be performed after type checking sequence in which the type tag of each argument is checked. If the types are not correct then error will be generated.

→ Perl and Prolog follow basically dynamically type checking

because data type of variables A+B in this case may be changed during program execution.

→ So the type checking must be done at run time.

**Advantage** of dynamic types is the flexibility in program design

→ No declaration are required
→ Type may be changed during execution.
→ programmer free from most concern about data types.
→

** Disadvantage :-

(i) **Difficult to Debug** :- We need to check program execution Paths for testing and in dynamic type checking, program execution paths for an operation is never checked

(ii) **Extra storage** :- Dynamic type checking need extra storage to keep type information during execution

(iii) **Seldom Hardware Support** :- As Hardware seldom Support the dynamic type checking so we have to implement in software which Reduces execution speed.

# Type checking

**Static Type checking :-** It is done at compile time.

Information needed at compile time is provided → by declaration
→ by language structures.

The information required includes

(i) **for Each operation :** The number, order, and datatypes of its arguments.

(ii) **For Each variables :** Name and data type of data object.

for example A+B In this types of A and B variables must not be changed.

(iii) **for Each Constant :-** Name and datatype & value.

**Eg :-** `Const int x = 28;`

`Const float x = 2.087;`

In this datatype, its value and name is specified and in further it checks value assigned should match its datatype.

**Advantages :-**

i) **Compiler Saves information :-** If the types of data is according to the operation

then Compiler Saves that information for checking Later Operations which further no need of Compilation.

(iii) **Checked Execution Paths:-** As Static type checking includes all operations that appears in any program Statement, all possible execution Paths are checked, and further testing for type errors is not needed.

So no type-tag on data objects at run-time are not required, and no dynamic checking is needed.

**Disadvantage :-** It affects many aspects of languages

(i) Declarations

(ii) Data Control Structures

(iii) provision of Compiling Separately Some Subprograms

## Strong Typing :-



If we can detect all types of errors statically in a program, we say that language is 'strongly typed.'

→ It provides a level of security to our program.

Eg:- $f : S \longrightarrow R$

In this function f with signature S generate output R and R is not out side the Range of R data type.

If every operation is type safe then automatically language is strongly typed.

Examples of Strongly typed languages are

C, Java, C++, Ruby Rail, Smalltalk, Python.



**Type Infer :-** In this, like in ML, the language implementation will infer any missing type information from other declared type.

Eg:-

```
Fun area(length:int, width : int):int =
    length * width;
```

This is the standard declaration which tells length and width of int data type and its return type is int and function name area.

but leaving any two of these declarations still leaves the function with only one interpretation.

Knowing that * can multiply together either two reals or two integers, ML interprets the following as equivalent to the previous example

```
Fun area(length, width)int  = length * width;
Fun area (length:int,width) = length * width;
Fun area ( length,width:int) = length * width;
```

However :-

```
Fun area(length, width) = length * width;
```

is invalid as it is now ambiguous as to the type of arguments. They could all be int or they could be real.