

# Type Checking

---

It is the activity of ensuring that the operands of an operator are of compatible types.

- Static :- Compile time.
- Dynamic :- Run-time.

Strongly typed language :

It is one in which each name in a program in the language has a single type associated with it, and that type is known at compile time (all types are statically bound).

- Problem :- Variables type may be known, but the storage location to which it is bound may store values of different types at different times.
- New definition :- It is one in which type errors are always detected.

# Strong Typing

---

## Examples:

- Not strongly typed:
  - FORTRAN :- Actual and formal parameters are not type checked; EQUIVALENCE statements.
  - C and C++ :- Both allow functions for which parameters are not type checked. Problem with *unions*.
  - Modula-2 :- Has type WORD, that avoids type checking. Problem with variant records.
- Nearly strongly typed:
  - Pascal :- Nearly strongly typed. Variant records.
  - Ada :- Type checking can be suspended by using function UNCHECKED\_CONVERSION.
- Strongly typed:
  - ML :- Statically known types.

## Type Compatibility

---

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

- Name :- Only if variables are in either the same declaration or in declarations that use the same name type; Very restrictive, easy to implement.

```
type indextype = 1..100; {a subrange type}
```

```
var
```

```
    count : integer;
```

```
    index : indextype;
```

- Structure :- Two variables have compatible types if their types have identical structures; Flexible but hard to implement.

```
type inch = real;
```

```
type cm = real;
```

Structurally equivalent but don't want to mix.

- C uses structural equivalence except for *struct*, for which it uses declaration equivalence.
- C++ uses name equivalence.
- Ada uses variant of name type compatibility.

# Primitive Data Types

---

Data types that are not defined in terms of other types.

- Numeric Types
  - Integer :– Different sizes; hardware support.
  - Floating-point :– Difficult to represent by finite number of binary digits.
  - Decimal :– Store fixed number of decimal digits.
- Boolean Types
  - Introduced in ALGOL 60. Range of values has 2 elements.
  - C and C++ uses numeric type variables.
- Character Types
  - Stored as numeric codings; ASCII.
- Pointer Types
  - Range of values – memory addresses and nil;  
*More later.*

## Structured Data Types

---

Constructed as an aggregation of other data types.

### Strings

- Primitive type :- FORTRAN 77, FORTRAN 90 and BASIC; Provide assignment, relational operators and catenation.
- Structured type :- Pascal, C, C++ and Ada; Single dimensional array of characters.
  - For C and C++, library functions provided whose header file is `string.h`.
  - Pascal has `packed` attribute.
  - Ada allows catenation, substring reference, comparison and assignment.
  - SNOBOL4 allows pattern matching operation.

### Length options

- Static length : Specified in declaration; Example
  - Pascal, Ada and FORTRAN.
- Limited dynamic length : Length varying upto fixed maximum; Does not require dynamic storage allocation; Example – C and C++.
- Dynamic length : Can have any size. Requires complex storage management; Example – SNOBOL4.

## User-Defined Ordinal Types

---

*Ordinal type* is one in which range of possible values can easily be associated with a set of positive integers.

Enumeration :- All possible values are enumerated.

Ex :- **type** soda is (coke, pepsi, dr. pepper)

*Same literal constant in more than one enum type ?*

**type** days is (SU, MO, TU, WE, TH, FR, SA);

**type** weekdy is (MO, TU, WE, TH, FR);

- Operations :- *pred, succ, position*
- Pascal, C and C++ – NO!
- Ada – YES!; Called **overloaded literals**.

Problem :- **for** day in TU .. TH **loop**

Solution :-

**for** day in weekdy(TU) .. weekdy(TH)**loop**

Subrange :- Continuous sequence of ordinal types.

Ex :- **type**

cardinal = 0..100;

Different from Ada's derived type.

Ex :- **type** DERIVED\_INT is new INTEGER;

**subtype** SUBRANGE\_INT is INTEGER;

DERIVED\_INT are *not* compatible with any INTEGER type.

## Arrays

---

Homogeneous aggregate of elements in which each element is identified by its position, relative to the first element.

A[I] Vs. A(I)

Why did a modern language like Ada choose A(I) ?

Lower bound of arrays are language dependent.

Different types of arrays : Based on binding of subscript value ranges and binding to storage.

- Static array :- Both are statically bound.
- Fixed stack-dynamic array :- Subscript ranges are statically bound but storage allocation dynamic (declaration elaboration time).
- Stack-dynamic array :- Both are dynamically bound (run-time).
- Heap-dynamic array :- Both are dynamic and change during the arrays lifetime.

Array Initialization :

- FORTRAN :- DATA statement.
- C and C++ :- `int list [] = {4, 5, 6, 7}`
- C and C++ :- `char name[] = "john";`
- Pascal and Mod-2 :- no initialization in declaration.
- Ada :- list in order or direct assignment.

## Array references

---

What about  $A[i, j]$ ?

First, we must agree to a storage scheme

*row-major order*

lay out as sequence of consecutive rows

rightmost subscript varies fastest

$A[1, 1], A[1, 2], A[1, 3], A[2, 1], A[2, 2], A[2, 3]$

*column-major order*

lay out as sequence of consecutive columns

leftmost subscript varies fastest

$A[1, 1], A[2, 1], A[1, 2], A[2, 2], A[1, 3], A[2, 3]$

*indirection vectors*       $A[v[i], w[j]]$

vector of pointers to pointers to ... to values

much more space

not amenable to analysis

## Array references

---

```
integer A[1:10];  
    ...  
x = A[i]
```

How do we compute the address of an array element?

$A[i]$

$$\textit{base} + (i - 1) \times w$$

where  $w$  is  $\textit{sizeof}(\textit{element})$

*in general:*  $\textit{base} + (i - \textit{low}) \times w$

*row-major order, two dimensions*

$$\textit{base} + ((i_1 - \textit{low}_1) \times (\textit{high}_2 - \textit{low}_2 + 1) + i_2 - \textit{low}_2) \times w$$

*column-major order, two dimensions*

$$\textit{base} + ((i_2 - \textit{low}_2) \times (\textit{high}_1 - \textit{low}_1 + 1) + i_1 - \textit{low}_1) \times w$$

This looks *expensive!*

## Array references

To minimize run-time costs, we need to know what they are

*On a typical RISC machine*

- integer add — 1 cycle
- integer loadi — 1 cycle
- integer load —  $\geq 1$  cycle (3–25)
- integer mult — 16 to 32 or more cycles

We should implement mult with shift whenever one argument is  $2^i$  and the other is unsigned

Element sizes are *always* in this form

*Of course, integer multiply via shift & add is often a win*

## Array addressing

The compiler should minimize the time spent in array addressing

*Several optimizations*

- pre-evaluation of subexpressions
- adopt a zero-based indexing scheme

*Consider  $A[i]$*

1.  $base + (i - low) \times w$
2.  $i \times w + base - low \times w$

*The second form is better*

- compile-time evaluable
- reference independent

## Array addressing

---

*the general address polynomial for row-major order*

$$((\dots(i_1 n_2 + i_2)n_3 + i_3)\dots)n_k + i_k) \times w$$

+base-

$$((\dots((low_1 \times n_2) + low_2)n_3 + low_3)\dots)n_k + low_k)$$

where  $n_i = high_i - low_i + 1$

*For fixed-size arrays,*

- final term is compile-time evaluable
- the  $n_i$  terms are compile-time evaluable

*Dope Vectors*

- save  $n_i$  and  $low_i$  and  $high_i$
- save constant part
- can be evaluated when declaration is seen